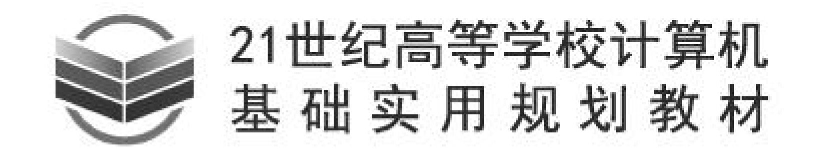


数据结构(Python版)

◎ 吕云翔 郭颖美 孟爻 编著

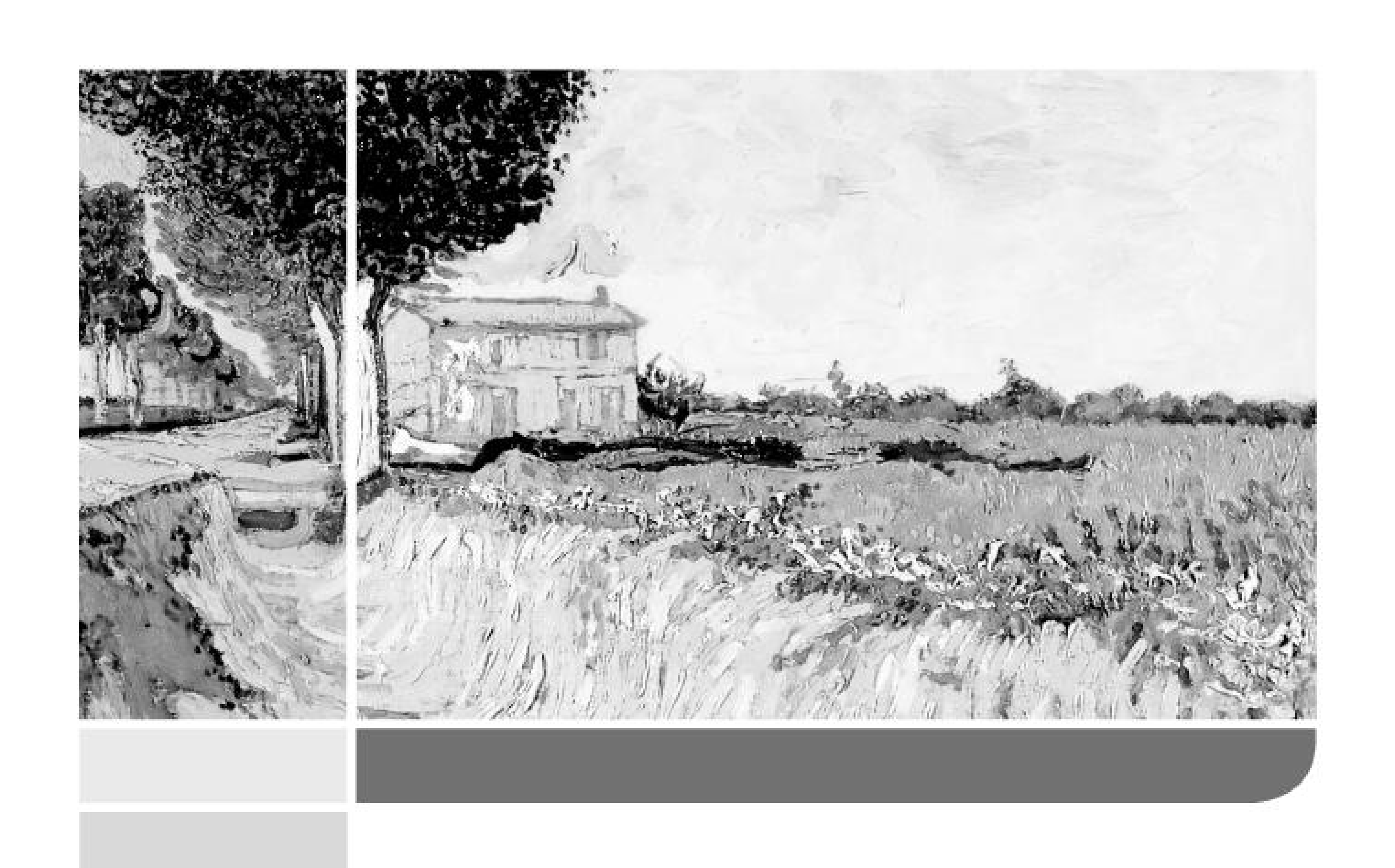



清华大学出版社



数据结构(Python版)

◎ 吕云翔 郭颖美 孟爻 编著



清華大学出版社
北京

内 容 简 介

本书在选材与编排上，贴近当前普通高等院校“数据结构”课程的现状和发展趋势，内容难易适度，突出实用性和应用性。本书并未面面俱到地介绍各种数据结构，而是通过分类和讲解典型结构，使读者对数据结构形成宏观认识。根据内容侧重，本书共分为8章，分别为绪论、线性表、栈和队列、串和数组、树结构、图、内排序和查找。

本书可以作为普通高校计算机相关专业“数据结构”课程的教材，也可以供学习数据结构的读者单独使用（包括参加计算机等级考试或相关专业自学考试）参考。

本书是高等院校计算机科学、软件工程及相关专业“数据结构”课程的理想教材，也可以供程序员、系统工程师等相关人员阅读参考。

图书在版编目（CIP）数据

数据结构：Python版/吕云翔，郭颖美，孟爻编著.—北京：清华大学出版社，2019
（21世纪高等学校计算机基础实用规划教材）
ISBN 978-7-302-52186-0

Ⅰ.①数… Ⅱ.①吕… ②郭… ③孟… Ⅲ.①软件工具－程序设计－数据结构－高等学校－教材 Ⅳ.①TP311.561 ②TP311.12

中国版本图书馆CIP数据核字（2019）第013064号

责任编辑：黄 芝
封面设计：刘 键
责任校对：胡伟民
责任印制：李红英

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 **邮 编：**100084
社 总 机：010-62770175 **邮 购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京国马印刷厂
经 销：全国新华书店
开 本：185mm×260mm **印 张：**14.25 **字 数：**356千字
版 次：2019年4月第1版 **印 次：**2019年4月第1次印刷
印 数：1～1500
定 价：39.50元

产品编号：080777-01

出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施"高等学校本科教学质量与教学改革工程（简称'质量工程'）"，通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部"质量工程"的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程（群）加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合"质量工程"的实施，满足各高校教学质量和教学改革的需要。

本系列教材立足于计算机公共课程领域，以公共基础课为主、专业基础课为辅，横向满足高校多层次教学的需要。在规划过程中体现了如下一些基本原则和特点。

（1）面向多层次、多学科专业，强调计算机在各专业中的应用。教材内容坚持基本理论适度，反映各层次对基本理论和原理的需求，同时加强实践和应用环节。

（2）反映教学需要，促进教学发展。教材要适应多样化的教学需要，正确把握教学内容和课程体系的改革方向，在选择教材内容和编写体系时注意体现素质教育、创新能力与实践能力的培养，为学生的知识、能力、素质协调发展创造条件。

（3）实施精品战略，突出重点，保证质量。规划教材把重点放在公共基础课和专业基础课的教材建设上；特别注意选择并安排一部分原来基础比较好的优秀教材或讲义修订再版，逐步形成精品教材；提倡并鼓励编写体现教学质量和教学改革成果的教材。

（4）主张一纲多本，合理配套。基础课和专业基础课教材配套，同一门课程可以有针对不同层次、面向不同专业的多本具有各自内容特点的教材。处理好教材统一性与多样化，基本教材与辅助教材、教学参考书，文字教材与软件教材的关系，实现教材系列资源配套。

(5) 依靠专家,择优选用。在制定教材规划时依靠各课程专家在调查研究本课程教材建设现状的基础上提出规划选题。在落实主编人选时,要引入竞争机制,通过申报、评审确定主题。书稿完成后要认真实行审稿程序,确保出书质量。

繁荣教材出版事业,提高教材质量的关键是教师。建立一支高水平教材编写梯队才能保证教材的编写质量和建设力度,希望有志于教材建设的教师能够加入到我们的编写队伍中来。

21世纪高等学校计算机基础实用规划教材

联系人:魏江江 weijj@tup.tsinghua.edu.cn

前　言

随着近年来计算概念的快速拓展，计算科学已经发展成为一个内涵繁杂的综合性学科，其至少可以划分为计算机工程(CE)、计算机科学(CS)、信息系统(IS)、信息技术(IT)和软件工程(SE)5 个领域，而且不同领域的人才所应具备的知识结构与能力侧重也不尽相同。尽管如此，数据结构在各领域的知识体系中仍然占据着重要的位置。“数据结构”是普通高等院校计算机专业和信息管理专业的一门必修课程，主要讨论数据的逻辑结构、在计算机中的储存结构以及对其进行的各种处理运算的方法和算法。

N. Wirth 早在 20 世纪 70 年代就指出“程序＝数据结构＋算法”。数据结构主要研究数据在计算机中储存、组织、传递和转换的过程及方法，这些也是构成与支撑算法的基础。近年来，随着面向对象技术的广泛应用，从数据结构的定义、分类、组成到设计、实现与分析的模式和方法都有了长足的发展，现代数据结构更加注重和强调数据结构的整体性、通用性、复用性、间接性和安全性。

基于上述情况，本书选择 Python 作为描述语言。Python 语言语法简洁优美，功能强大，有着广泛的应用领域，如互联网、大数据、人工智能等领域。因此，学习 Python 语言，在未来的学习和工作中，都有用武之地。同时，Python 语言相对于大多数高级语言，更加适合初学者学习，Python 的语法与伪代码描述很相似，逻辑清晰；此外，Python 语言也同样具有大部分高级语言的特性，对计算机相关专业的学生未来学习其他编程语言有所帮助。

在内容的选取与结构安排上，本书通过分类和讲解典型结构使读者对数据结构形成宏观认识。根据内容的侧重，本书分 8 章，分别为绪论、线性表、栈和队列、串和数组、树结构、图、排序和查找。

第 1 章介绍数据结构的基本概念、算法描述、算法的时间复杂度和空间复杂度等内容。本章是全书的基础。

第 2 章主要介绍线性表的基本概念和抽象数据类型的定义，线性表的顺序和链式两种存储方式的标识，以及线性表的基本操作实现和相应应用。

第 3 章简要介绍栈和队列的基本概念和抽象数据类型定义，栈和队列在顺序存储和链式存储结构下的基本操作和应用。

第 4 章主要介绍串的基本概念和数据类型定义，串的存储结构、基本操作实现和应用等内容。

第 5 章主要介绍树和二叉树的基本概念，详细介绍二叉树的性质和存储结构、便利方法的实现及应用、哈夫曼树的概念和构造方法。

第 6 章主要介绍图的基本概念、抽象数据类型定义、存储结构和遍历方法，还介绍最小生成树的基本概念和方法、最短路径的相关算法、拓扑排序的概念和实现方法。

第7章介绍排序的基本概念,插入排序、交换排序、选择排序、归并排序等多种排序的原理、实现方法及性能分析。

第8章主要介绍查找的基本概念,顺序查找、二分查找等查找的原理、实现方法和性能分析,平衡二叉树、哈希表的概念、结构定义和实现方法。

本书的理论知识的教学安排建议如下表所示。

章　节	内　容	学时数
第1章	绪论	2
第2章	线性表	4～6
第3章	栈和队列	6～8
第4章	串和数组	2～4
第5章	树结构	6～8
第6章	图	4～8
第7章	排序	4～6
第8章	查找	4～6

建议先修课程:Python语言。

建议理论教学时数:32～48学时。

建议实验(实践)教学时数:16～32学时。

本书中的所有算法都已经通过上机调试,尽量保证算法的正确性。在每章内容后都有小结,便于读者复习总结,并配有丰富的习题,包括选择题、填空题、算法设计题等,给读者更多的思考空间。

本书在以下几个方面具有突出特色。

(1)内容精炼,强化基础,合理安排内容结构,做到深入浅出、循序渐进。

本书各章节都从基本概念入手,逐步介绍其特点和基本操作的实现,把重点放在基础知识的介绍上,缩减难度较大的内容,使理论叙述简洁明了、重点突出、详略得当。

(2)应用实例丰富、完整。

本书通过丰富的应用实例和源代码使理论和应用紧密结合,增强学生的理解能力,锻炼程序设计思维。代码有详细明了的注释,易于阅读。

(3)每章后面附有小结和习题,便于学习、总结和提高。

本书结合学生的学习实际选择难度适中、逻辑合理、适于初学者和进阶者开拓思路、深入了解数据结构使用方法和技巧的习题,并附有详细的解答过程和注意要点,达到通俗易懂、深入浅出的效果,培养读者迁移知识的能力。

(4)采用Python抽象类体现方法的通用性。

本书采用面向对象的观点讨论数据结构技术,先将抽象数据类型定义成接口,再结合具体的存储结构加以实现,并以各实现类为线索对类中各种操作的实现方法加以说明。

(5)图文并茂,便于学生直观地理解数据结构与算法。

本书通过图表的方式对数据结构及相应操作进行简单直接的描述,使内容更加浅显易懂。

教师可以按照自己对数据结构的理解适当地跳过一些章节,也可以根据教学目标灵活

地调整章节的顺序，增减各章的学时数。

本书的作者为吕云翔、郭颖美、孟爻，曾洪立、吕彼佳、姜彦华参与了部分内容的编写并进行了素材整理及配套资源制作等。

由于数据结构本身还在探索之中，加上我们的水平和能力有限，本书难免有疏漏之处，恳请各位同仁和广大读者给予批评指正，也希望各位能将实践过程中的经验和心得与我们交流(yunxianglu@hotmail.com)。

目　录

第 1 章　绪论 …… 1

1.1　引言 …… 1

1.1.1　学习目的 …… 1

1.1.2　课程内容 …… 1

1.2　基本概念 …… 2

1.2.1　数据与数据结构 …… 2

1.2.2　数据类型与抽象数据类型 …… 5

1.3　算法 …… 7

1.3.1　算法的概念 …… 7

1.3.2　算法描述 …… 8

1.3.3　算法分析 …… 9

小结 …… 12

习题 1 …… 12

第 2 章　线性表 …… 15

2.1　线性表及其基本操作 …… 15

2.1.1　线性表的基本概念 …… 15

2.1.2　抽象数据类型描述 …… 15

2.1.3　线性表的存储和实现 …… 16

2.2　线性表的顺序存储 …… 17

2.2.1　顺序表 …… 17

2.2.2　顺序表的基本操作实现 …… 18

2.3　线性表的链式存储和实现 …… 22

2.3.1　单链表 …… 22

2.3.2　单链表的基本操作实现 …… 24

2.3.3　其他链表 …… 27

2.4　顺序表与链表的比较 …… 28

小结 …… 28

习题 2 …… 29

第 3 章　栈和队列 ………………………………………………………………………… 31
3.1　栈………………………………………………………………………………………… 31
3.1.1　栈的基本概念 …………………………………………………………………… 31
3.1.2　栈的抽象数据类型描述 ………………………………………………………… 31
3.1.3　顺序栈 …………………………………………………………………………… 32
3.1.4　链栈 ……………………………………………………………………………… 34
3.2　队列……………………………………………………………………………………… 37
3.2.1　队列的基本概念 ………………………………………………………………… 37
3.2.2　队列的抽象数据类型描述 ……………………………………………………… 38
3.2.3　顺序队列 ………………………………………………………………………… 38
3.2.4　链队列 …………………………………………………………………………… 42
3.2.5　优先级队列 ……………………………………………………………………… 44
3.3　栈和队列的比较………………………………………………………………………… 46
小结 ………………………………………………………………………………………… 46
习题 3 ……………………………………………………………………………………… 47

第 4 章　串和数组 ………………………………………………………………………… 50
4.1　串………………………………………………………………………………………… 50
4.1.1　串的基本概念 …………………………………………………………………… 50
4.1.2　串的抽象数据类型描述 ………………………………………………………… 50
4.1.3　顺序串 …………………………………………………………………………… 51
4.1.4　链串 ……………………………………………………………………………… 55
4.2　串的模式匹配…………………………………………………………………………… 56
4.2.1　Brute Force 算法 ………………………………………………………………… 56
4.2.2　KMP 算法 ………………………………………………………………………… 56
4.3　数组……………………………………………………………………………………… 60
4.3.1　数组的基本概念 ………………………………………………………………… 60
4.3.2　数组的特性 ……………………………………………………………………… 60
4.3.3　数组的遍历 ……………………………………………………………………… 61
4.4　特殊矩阵的压缩存储…………………………………………………………………… 61
4.4.1　三角矩阵的压缩存储 …………………………………………………………… 62
4.4.2　对称矩阵的压缩存储 …………………………………………………………… 62
4.4.3　对角矩阵的压缩存储 …………………………………………………………… 63
4.4.4　稀疏矩阵的压缩存储 …………………………………………………………… 63
小结 ………………………………………………………………………………………… 65
习题 4 ……………………………………………………………………………………… 65

第 5 章　树结构 …… 68

5.1　树 …… 68
5.1.1　树的基本概念 …… 68
5.1.2　树的术语 …… 68
5.2　二叉树 …… 69
5.2.1　二叉树的基本概念 …… 69
5.2.2　二叉树的性质 …… 70
5.2.3　二叉树的存储结构 …… 71
5.2.4　二叉树的遍历 …… 72
5.2.5　二叉树遍历算法的应用 …… 76
5.2.6　二叉树的建立 …… 77
5.3　哈夫曼树及哈夫曼编码 …… 79
5.3.1　哈夫曼树的基本概念 …… 80
5.3.2　哈夫曼树的构造 …… 80
5.3.3　哈夫曼编码 …… 80
5.3.4　构造哈夫曼树和哈夫曼编码的类的描述 …… 81
5.4　树和森林 …… 83
5.4.1　树的存储结构 …… 83
5.4.2　树的遍历规则 …… 84
小结 …… 84
习题 5 …… 84

第 6 章　图 …… 87

6.1　图概述 …… 87
6.1.1　图的基本概念 …… 87
6.1.2　图的抽象数据类型描述 …… 89
6.2　图的存储结构 …… 89
6.2.1　邻接矩阵 …… 90
6.2.2　邻接表 …… 93
6.3　图的遍历 …… 97
6.4　最小生成树 …… 102
6.4.1　最小生成树的基本概念 …… 102
6.4.2　Kruskal 算法 …… 102
6.4.3　Prim 算法 …… 102
6.5　最短路径 …… 104
6.5.1　单源最短路径 …… 104
6.5.2　求任意两个顶点间的最短路径 …… 106
6.6　拓扑排序和关键路径 …… 107

6.6.1 拓扑排序 …… 107
6.6.2 关键路径 …… 108
小结 …… 110
习题 6 …… 111

第 7 章 排序 …… 113

7.1 排序概述 …… 113
7.1.1 排序的基本概念 …… 113
7.1.2 排序算法的性能评价 …… 113
7.1.3 待排序的记录和顺序表的类描述 …… 113
7.2 插入排序 …… 114
7.2.1 直接插入排序 …… 114
7.2.2 希尔排序 …… 115
7.3 交换排序 …… 116
7.3.1 冒泡排序 …… 116
7.3.2 快速排序 …… 117
7.4 选择排序 …… 120
7.4.1 直接选择排序 …… 120
7.4.2 堆排序 …… 121
7.5 归并排序 …… 123
小结 …… 127
习题 7 …… 128

第 8 章 查找 …… 130

8.1 查找的基本概念 …… 130
8.1.1 什么是查找 …… 130
8.1.2 查找表 …… 130
8.1.3 平均查找长度 …… 130
8.2 静态表查找 …… 131
8.2.1 顺序查找 …… 131
8.2.2 二分查找 …… 131
8.2.3 分块查找 …… 132
8.3 动态表查找 …… 133
8.3.1 二叉排序树查找 …… 133
8.3.2 平衡二叉树 …… 136
8.3.3 B^- 树和 B^+ 树 …… 138
8.4 哈希表查找 …… 139
8.4.1 哈希表的概念 …… 139
8.4.2 哈希函数 …… 139

8.4.3 解决冲突的方法…… 140
8.4.4 哈希表查找性能分析…… 142
小结…… 144
习题 8 …… 145

附录 A 数据结构试卷…… 147

数据结构试卷(一)…… 147
数据结构试卷(二)…… 150
数据结构试卷(三)…… 152
数据结构试卷(四)…… 155
数据结构试卷(五)…… 157

附录 B 实践题 …… 160

第 2 章 线性表…… 160
第 3 章 栈和队列…… 169
第 4 章 串和数组…… 174
第 5 章 树结构…… 181
第 6 章 图…… 188
第 7 章 排序…… 201
第 8 章 查找…… 205

参考文献…… 210

第1章 绪　论

1.1 引　言

1.1.1 学习目的

软件设计是计算机学科的核心内容之一,如何有效地组织数据和处理数据是软件设计的基本内容,直接关系软件的运行效率和工程化程度。

数据结构是计算机科学中的一门综合性的专业基础课,和数学、计算机硬件、软件等课程有着十分密切的关系,是软件设计的重要理论和实践基础。数据结构是一门理论与实践并重的课程,学生既要掌握数据结构的基础理论知识,又要掌握运行和调试程序的基本技能,因此,数据结构课程是培养学生的程序设计能力的必不可少的重要环节。

在计算机发展的初期,计算机处理的对象多为简单的数值数据。由于早期所涉及的运算对象是简单的整型、实型或布尔类型数据,数据量小且结构简单,所以程序设计者的主要精力集中在程序设计的技巧上。而现在,随着计算机和信息技术的飞速发展,计算机应用远远超出了单纯进行数值计算的范畴,从早期的科学计算扩大到过程控制、管理和数据处理等领域。处理非数值计算性问题占用了90%以上的机器时间,涉及了更为复杂的数据结构和数据元素间的相互关系。因此,数学分析和算法不再是解决这类问题的关键,只有设计出合适的数据结构才能有效地解决问题。

使用计算机解决实际的应用问题一般需要经过以下3个步骤。

(1) 从具体问题中抽象出适当的数学模型:分析问题,提取操作的对象,找出操作对象之间的逻辑关系,给出相应的数学模型。

(2) 设计解决此数学模型的算法。

(3) 编程、运行、调试,得出结果。

1.1.2 课程内容

数据结构课程主要讨论软件开发过程中的设计阶段,也涉及分析和编码阶段的若干问题。主要通过以下3个步骤用计算机解决问题。

(1) 抽象求解问题中需处理的数据对象的逻辑结构。

(2) 根据求解问题需要完成的功能特性实现存储结构表述。

(3) 确定为求解问题而需要进行的操作或运算。

为了构造和实现出好的数据结构,必须将以上三者结合,充分考虑与各种典型的逻辑结构、存储结构、数据结构相关的操作和实现及实现方法的性能,因此课程内容可归纳为如

表1.1所示。

表1.1 “数据结构”课程

过程	方面	
	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	储存结构	算法
评价	不同数据结构比较和算法性能分析	

1.2 基本概念

1.2.1 数据与数据结构

1. 数据

数据(Data)是能够被计算机程序识别、存储、加工和处理的描述客观事物的数字等符号集合的总称。数据是信息的载体,是计算机程序处理对象的集合,也是计算机处理信息的某种特定的符号化表示形式,除了整数、实数等数值数据外,还包括字符串等非数值数据及图形、图像、音频、视频等多媒体数据。如表1.2所示,书籍信息表中所含的数据就是表中所有书籍记录的集合。

表1.2 书籍信息表

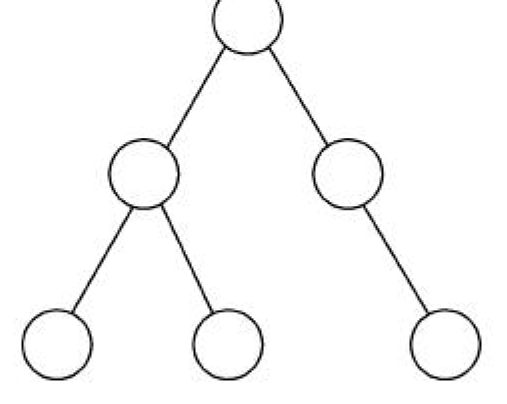
书名	作者	出版社	价格
软件工程实用教程	吕云翔	清华大学出版社	49.00

2. 数据项

数据项(Data Item)是具有独立含义的、数据不可分割的最小标识单位,是数据元素的组成部分,也可称为字段和域。如表1.2所示,“书名”“作者”“出版社”“价格”都是数据项,数据项可分为两种:一种为简单数据项,是进行数据处理时不能分割的最小单位,如“书名”“价格”;另一种为组合数据项,可以划分为更小的项,如“作者”可以划分为“第一作者”“第二作者”。在一个数据元素中能够识别该元素的一个或者多个数据项称为关键字。

3. 数据元素

数据元素(Data Element)是数据的基本单位,又可称为元素、节点、顶点和记录,是一个数据整体中可以标识和访问的数据单元。如表1.2中的一行数据称为一个数据元素或一条记录。在图或树中数据元素用圆圈表示,如图1.1所示,每个圆圈都代表一个数据元素,称为一个顶点。


图1.1 结构图

一个数据元素可以是不可分割的原子项,也可由若干数据项组成。如表1.2所示,书籍信息表中的每一条书籍记录就是一个数据元素,它由书名、作者、出版社、价格等数据项构成。

4. 数据对象

数据对象(Data Object)是性质相同的数据元素的集合,也叫数据元素类,是数据的一个子集,数据元素是数据对象的一个实例。如表 1.2 中所有出版社为清华大学出版社的书籍记录可组成一个数据对象,第一行的书籍记录则为该数据对象的一个实例。

5. 数据结构

数据结构(Data Structure)是相互之间存在着一种或者多种关系的数据元素的集合,数据结构概念包含 3 个方面的内容,即数据的逻辑结构、数据的存储结构和数据操作,只有 3 个方面的内容相同才能称为完全相同的数据结构。

1) 逻辑结构

数据的逻辑结构是指数据元素之间存在的逻辑关系,由数据元素的集合和定义在此集合上的关系组成。数据的逻辑结构与数据的存储无关,独立于计算机,是从具体问题抽象出来的数学模型。数据的逻辑结构由两个要素构成:一个是数据元素的集合;另一个是关系的集合。

根据数据元素间逻辑关系的不同特性,数据的逻辑结构可分为以下 4 类。

(1) 集合:集合中元素的关系极为松散,关系为“属于同一个集合”。集合的逻辑结构如图 1.2(a)所示。

(2) 线性结构:线性结构是数据元素中具有线性关系的数据结构,线性结构中的节点存在“一对一”的关系。开始节点和终端节点都是唯一的,除开始节点和终端节点外,每个节点有且仅有一个前驱节点和一个后继节点,开始节点仅有一个后继节点,终端节点仅有一个前驱节点。整数序列、字母表都是线性结构。线性结构的逻辑结构如图 1.2(b)所示。

(3) 树结构:树结构是数据元素之间具有层次关系的一种非线性结构,树结构中的节点存在“一对多”的关系。除根节点外,每个节点有且仅有一个前驱节点,所有节点可以有零个或者多个后继节点,家谱、Windows 文件系统的组织方式、淘汰赛的比赛结果都是树结构。树结构的逻辑结构如图 1.2(c)所示。

(4) 图形结构:图形结构也是一种非线性结构,图形结构中的节点存在“多对多”的关系。所有节点都可以有多个前驱节点和后继节点。交通图、飞机航班路线图都是图形结构。图形结构的逻辑结构如图 1.2(d)所示。

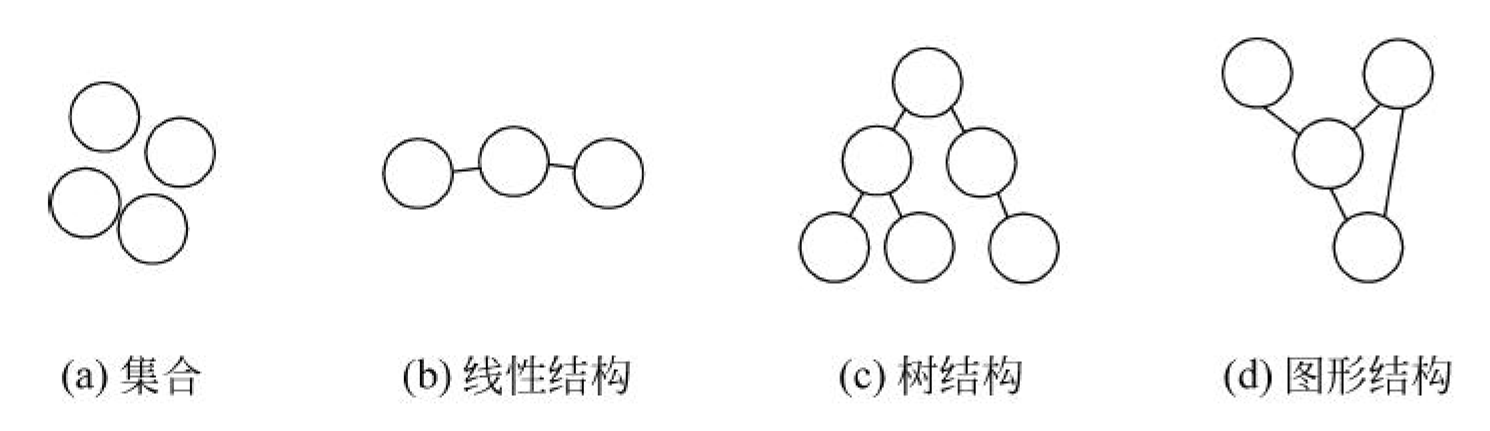


图 1.2 逻辑结构图

数据的逻辑结构涉及两个方面的内容,一是数据元素,二是数据元素间的逻辑关系,所以可以采用一个二元组来定义数据的逻辑结构:

$$\text{Logica_Structures}=(\mathrm{D},\mathrm{R})$$

其中,D 是数据元素的集合,R 是数据元素间逻辑关系的集合。若 a_1 和 a_2 都属于 D,并且 $<a_1,a_2>\in R$,则称 a_1 是 a_2 的前驱元素,a_2 是 a_1 的后继元素。一般情况下,若 $R_1\in R$,

则 R_1 是 $D\times D$ 的关系子集。

【例 1.1】 根据给出的数据对象和数据关系求解相应的逻辑结构。

(1) 设数据对象 $D=\{1,2,3,4,5,6\}$,数据关系 $R=\{<1,2>,<1,3>,<1,4>,<3,5>,<3,6>\}$,试画出它们对应的逻辑图形表示,并指出它们属于何种逻辑结构。

解: 该题中数据元素间的关系是一种一对多的关系,其中只有一个元素没有直接前驱,其余元素有且仅有一个直接前驱,而元素的直接后继可以有一个或多个,也可以没有。

其逻辑图形表示如图 1.3 所示,属树结构。

(2) 设数据对象 $D=\{1,2,3,4,5\}$,数据关系 $R=\{<1,2>,<2,3>,<5,1>,<2,5>,<4,1>,<4,5>,<4,3>\}$,试画出它们对应的逻辑图形表示,并指出它们属于何种逻辑结构。

解: 该题中数据元素间的关系是多对多的关系,元素有多个直接前驱和直接后继,也可以没有。

其逻辑图形表示如图 1.4 所示,属图形结构。

(3) 设数据对象 $D=\{1,2,3,4,5,6\}$,数据关系 $R=\{<1,2>,<1,3>,<3,4>,<3,6>,<4,5>\}$,试画出它们对应的逻辑图形表示,并指出它们属于何种逻辑结构。

解: 该题中数据元素间的关系是一种一对多的关系,其中只有一个元素没有直接前驱,其余元素有且仅有一个直接前驱,而元素的直接后继可以有一个或多个,也可以没有。

其逻辑图形表示如图 1.5 所示,属树结构。

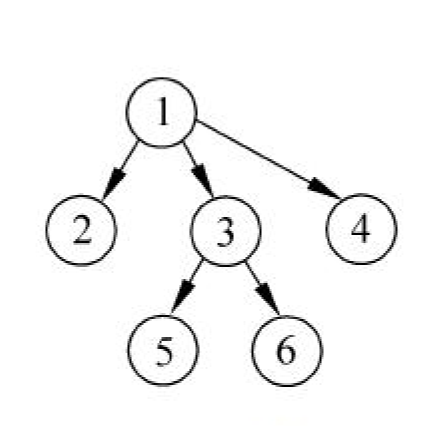


图 1.3 树结构

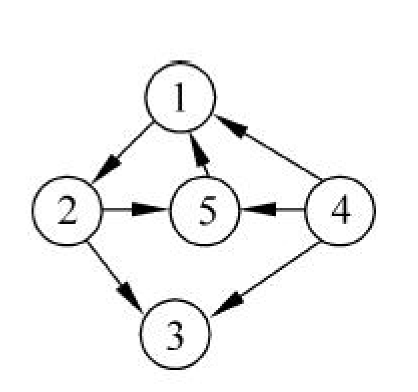


图 1.4 图形结构

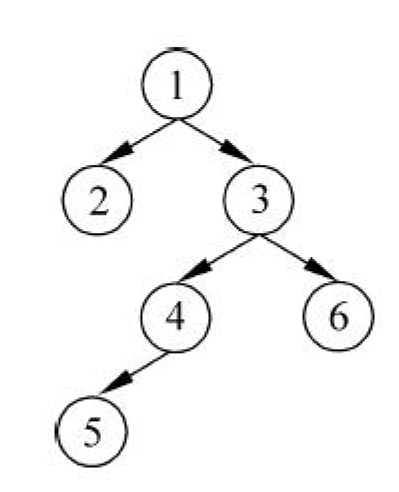


图 1.5 树结构

2) 存储结构

逻辑结构在计算机中的存储表示或实现叫数据的存储结构,也叫物理结构。数据的逻辑结构从逻辑关系角度观察数据,与数据的存储无关,是独立于计算机的;而数据的存储结构是逻辑结构在计算机中的实现,依赖于计算机。

数据的存储结构可以分为以下 4 类。

(1) 顺序存储结构:顺序存储结构在连续的存储单元中存放数据元素,元素的物理存储次序和逻辑次序一致,即物理位置相邻的元素在逻辑上也相邻,每个元素与其前驱元素和后继元素的存储位置相邻,数据元素的物理存储结构体现它们之间的逻辑关系。顺序存储结构可通过程序设计语言的数组实现。

(2) 链式存储结构:链式存储结构使用地址分散的存储单元存放数据元素,逻辑上相邻的数据元素的物理位置不一定相邻,数据元素间的逻辑关系通常由附加的指针表示,指针记录前驱元素和后继元素的存储地址。数据元素由数据元素值和存放逻辑关系的指针共同构成,通过指针将相互直接关联的节点链接起来,节点间的链接关系体现数据元素之间的逻辑关系。

(3) 索引存储结构：索引存储结构在存储数据元素的基础上增加索引表。索引表的项由关键字和地址构成，其中关键字唯一标识一个数据元素，地址为该数据元素存储地址的首地址。

(4) 散列存储结构：散列存储结构也叫哈希存储结构，数据元素的具体存储地址根据该数据元素的关键字值通过散列函数直接计算出来。

顺序存储结构和链式存储结构是两种最基本、最常用的存储结构。在实际应用中可以将顺序存储结构和链式存储结构进行组合构造出复杂的存储结构，根据所处理问题的实际情况选择合适的存储结构，达到操作简单、高效的目的。

3) 数据操作

数据操作是指对数据结构中的数据元素进行运算或处理。数据操作定义在数据的逻辑结构上，每种逻辑结构都需要一组对其数据元素进行处理以实现特定功能的操作，如插入、删除、更新等。数据操作的实现依赖于数据的存储结构。

常用的数据操作有以下 7 种。

(1) 创建操作。

(2) 插入操作。

(3) 删除操作。

(4) 查找操作。

(5) 修改操作。

(6) 遍历操作。

(7) 销毁操作。

1.2.2 数据类型与抽象数据类型

1. 数据类型

数据类型(Data Type)是一组性质相同的值的集合和定义在此集合上的一组操作的总称。在用高级程序语言编写的程序中必须对程序中出现的每个变量、常量明确说明它们所属的数据类型。确定数据的类型意味着确定了数据的性质以及对数据进行的运算和操作，同时数据也受到类型的保护，确保对数据不能进行非法操作。不同类型的变量的取值范围不同，所能进行的操作不同。例如 Python 语言中的 32 位整数类型 int 利用 32 位补码进行存储表示，取值范围为[-2^31,…,-2,-1,0,1,2,…,2^31-1]，可以进行的操作集合为[+,-,*,/,=,==,!=,<,>=,>,<=]。

高级程序设计语言通常预定义基本数据类型和构造数据类型。基本数据类型是只能作为一个整体来进行处理不可分解的数据类型。Python 语言的基本数据类型有数字(Number)、字符串(String)、列表(List)、元组(Tuple)、字典(Dictionary)及其包含的细分类型。构造数据类型是使用已有的基本数据类型和已定义的构造数据类型通过一定的语法规则组织起来的数据类型。在 Python 中通常通过类的声明引入新的数据类型。类的对象是新的数据类型的实例，类的成员变量确定数据表示方法和存储结构，类的函数确定数据可以进行的操作。

2. 数据抽象和抽象数据类型

1) 数据抽象

数据抽象是指“定义和实现相分离”，即将一个类型的数据及其上的操作的逻辑含义和

具体实现相分离,只考虑执行什么操作(做什么),而不考虑怎样实现这些操作(怎样做)。比如,程序设计语言中的数据类型是抽象的,仅描述数据的特性和对数据操作的语法规则,并没有说明这些数据类型是如何实现的,程序员使用数据类型只需要按照语法规则考虑对数据执行什么操作,而不必考虑怎样实现这些操作。

数据抽象是一种信息隐蔽技术,可利用数据抽象研究复杂对象,忽略次要和实现细节,抽象出本质特征,抽象层次越高,复用程度越高。数据抽象是通过抽象数据类型来实现的。

2) 抽象数据类型

抽象数据类型(Abstract Data Type,ADT)是从问题的数学模型中抽象出来的逻辑结构及定义在逻辑结构上的一组操作,仅描述了数据的特性和数据操作的语法规则,隐藏了数据的存储结构和操作的实现细节。

抽象数据类型是实现软件模块化设计思想的重要手段,一个抽象数据类型是描述一种特定功能的基本模块,由各种基本模块可组织和构造起来一个大型的软件系统。

在一般的面向对象语言中,抽象数据类型通常可以采用抽象类或接口的方式进行描述。Python 语言本身没有提供类似其他面向对象语言的抽象类、接口语法,但我们仍可以在 Python 中引用 abc 模块来实现抽象类。

【例 1.2】 用 Python 语言 abc 模块实现抽象类描述集合这一抽象数据类型。

(1) 声明抽象类。

```
from abc import ABCMeta,abstractproperty,abstractmethod

class Set(metaclass = ABCMeta):
    '''
    集合抽象类,metaclass = ABCMeta 表示将 Set 类作为 ABCMeta 的子类
    继承于 abc.ABCMeta 的类可以使用 abstractproperty,abstractmethod 修饰器声明虚属性与虚方法
    '''
    @abstractproperty
    def size(self):
        '''
        返回集合中元素的个数
        '''
        pass
    @abstractmethod
    def isEmpty(self):
        '''
        判断集合是否为空
        '''
        pass
    @abstractmethod
    def search(self,key):
        '''
        在集合中查找关键字为 key 的元素并返回
        '''
        pass
    @abstractmethod
    def contains(self,x):
```

```
        '''
        判断集合中是否包含元素 x
        '''
        pass
    @abstractmethod
    def add(self,x):
        '''
        向集合中添加元素 x
        '''
        pass
    @abstractmethod
    def remove(self,key):
        '''
        删除集合中关键字值为 key 的元素
        '''
        pass
    @abstractmethod
    def clear(self):
        '''
        删除集合中的所有元素
        '''
        pass
```

(2) 声明实现抽象类的类。

```
class HashSet(Set):
    @property
    def size(self):
        pass
    def isEmpty(self):
        pass
    def search(self,key):
        pass
    def contains(self,x):
        pass
    def add(self,x):
        pass
    def remove(self,key):
        pass
    def clear(self):
        pass
    def __init__(self):
        pass
```

1.3 算　　法

1.3.1 算法的概念

1. 算法的定义

算法是有穷规则的集合,其规则确定一个解决某一特定类型问题的指令序列,其中每一

条指令表示计算机的一个或者多个操作。

算法必须满足以下 5 个特性。

(1) 有穷性：对于任意的合法输入值，算法必须在执行有穷步骤后结束，并且每一步都在有穷的时间内完成。

(2) 确定性：算法对各种情况下执行的每个操作都有确切的规定，算法的执行者和阅读者都能明确其含义和如何执行，并且在任何条件下算法都只有一条执行路径。

(3) 可行性：算法中的操作必须都可以通过已经实现的基本操作运算有限次实现，并且每一条指令都符合语法规则，满足语义要求，都能被确切执行。

(4) 有输入：输入数据是算法的处理对象，一个算法具有零个或多个输入数据，既可以由算法指定，也可以在算法执行过程中通过输入得到。

(5) 有输出：输出数据是算法对输入数据进行信息加工后得到的结果，输出数据和输入数据具有确定的对应关系，即算法的功能。一个算法有一个或多个输出数据。

2. 算法设计的目标

算法设计应该满足以下 4 个基本目标。

(1) 正确性：算法应满足应用问题的需求，这是算法设计最重要、最基本的目标。

(2) 健壮性：算法应具有良好的容错性，可以检查错误是否出现并且对错误进行适当的处理，即使输入的数据不合适，也能避免出现不可控的结果。

(3) 高效率：算法的执行时间越短，时间效率越高；算法执行时所占的存储空间越小，空间效率越高。时间效率和空间效率往往不可兼得，用户在解决实际问题时要根据实际情况权衡得失，进行高效率算法的设计。

(4) 可读性：算法的表达思路应清晰，层次分明，易于理解，可读性强，以便于后续对算法的使用和修改。

3. 算法与数据结构

算法建立在数据结构之上，对数据结构的操作需要使用算法来描述。算法设计依赖于数据的逻辑结构，算法实现依赖于数据的存储结构。

1.3.2 算法描述

算法可以采用多种语言进行描述，主要分为以下 3 种。

(1) 自然语言：自然语言用中文或英文对算法进行表达，简单易懂，但缺乏严谨性。如对顺序查找算法进行自然语言描述，在表 1.3 所示的学生信息表中以学号为关键字进行顺序查找，从线性表的一端开始依次比较学生的学号和所给定值，当学生的学号与所给定值相等时查找成功，查找操作结束；否则继续比较，直到比较完所有元素，查找失败，查找操作结束。

表 1.3 学生信息表

学号	姓名	性别	年龄	班级
001	张三	女	19	1
002	李四	男	19	1
003	王五	男	19	1

(2) 程序设计语言：使用某种具体的程序设计语言(如 Python 语言)对算法进行描述。此种方式严谨,算法可直接在计算机上执行,但算法复杂、不易理解,需要借助大量的外部注释才能使用户明白。例如：

```
def seqSearch(int key):
    i = 0,n = self.length()
    while i < n and r[i].id != key:
        i += 1
    if i < n:
        return i # 查找成功,返回标记序号
    else:
        return -1 # 查找失败
```

(3) 伪代码：伪代码是介于自然语言和程序设计语言之间的算法描述语言,是将程序设计语言的语法规则用自然语言进行表示,忽略了严格的语法规则和描述细节,更易被用户理解,并且更容易转换成程序设计语言执行。例如：

```
key = 001
for 学号 in 学生信息表:
    if 学号 == key:
        查找成功,结束
查找失败,结束
```

【例 1.3】 设计算法求两个整数的最大公约数。

解：求两个整数的最大公约数有 3 种方法。

(1) 质因数分解法。

假设 c 为两个整数 a 和 b 的最大公约数。用数学方法求两个数的最大公约数,分别将 a 和 b 两个整数分解成若干质因数的乘积,再从中选择最大的公约数。此种方法很难用于实际计算之中,因为大数的质因数很难进行分解。

(2) 更相减损术。

中国古代的数学经典著作《九章算术》中写道："以少减多,更相减损,求其等也,以等数约之,即除也,其所以相减者皆等数之重叠,顾以等数约之。"其中,等数即指两数的最大公约数。

(3) 辗转相除法。

实际上,辗转相除法就是现代版的更相减损术,使用循环实现：

```
def gcd(a,b):
    while b != 0 :
        tmp = a % b
        a = b
        b = tmp
    return a
```

1.3.3 算法分析

如果要解决一个实际问题,经常有多个算法可以选择,每个算法都有其自身的优缺点,

为了选择合适的算法,需要利用算法分析技术评价算法的效率。算法分析技术主要是通过某种方法讨论算法的复杂度,评价算法的效率,以便在解决实际问题时根据实际情况和算法的优缺点对算法进行取舍。算法的优劣主要通过算法复杂度进行衡量,复杂度的高低反映了所需计算机资源的多少。计算机资源主要包括时间资源和空间资源。因此算法的复杂度通常以时间复杂度和空间复杂度来体现。在解决实际问题时优先选择复杂度较低的算法。

1. 算法的时间复杂度

算法的时间复杂度(Time Complexity)是指算法的执行时间随问题规模的变化而变化的趋势,反映算法执行时间的长短。

算法的执行时间是用算法编写的程序在计算机上运行的时间,它是算法中涉及的所有基本运算的执行时间之和。执行时间依赖于计算机的软/硬件系统,如处理器速度、程序运行的软件环境、编写程序采用的计算机语言、编译产生的机器语言代码等,因此不能使用真实的绝对时间来表示算法的效率,而应只考虑算法执行时间和问题规模之间的关系。

当问题的规模为 n 时,$T(n)$表示此时算法的执行时间,称为算法的时间复杂度,当 n 增大时,$T(n)$也随之增大。

假设一个算法是由 n 条指令序列构成的集合,算法的执行时间如下:

$$T(n) = \sum_{i=1}^{n} \text{指令序列}(i)\text{的执行次数} \times \text{指令序列}(i)\text{的执行时间}$$

由于算法的时间复杂度表示算法执行时间随数据规模的增大而增大的趋势,并非绝对时间,且与指令序列的执行次数成正比,所以可通过计算算法中指令序列的执行次数之和来估算一个算法的执行时间。显然在一个算法中指令序列的执行次数越少,其运行时间也越少;指令序列的执行次数越多,其运行时间也越多。

通常采用算法的渐进分析中的大写的 O 表示法作为算法时间复杂度的渐进度量值,称为算法的渐进时间复杂度。大写的 O 表示法是指当且仅当存在正整数 c 和 n_0,使得 $0 \leqslant T(n) \leqslant cf(n)$对所有的 $n \geqslant n_0$ 成立时,算法执行时间的增长率与 $f(n)$的增长率相同,记为$T(n)=O(f(n))$。

一般地,如果 $f(n)=a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n^1 + a_0$,且 $a_i \geqslant 0$,$T(n)=O(n^k)$,即使用大 O 表示法时只需保留关于数据元素个数 n 的多项式的最高次幂的项并去掉其系数。比如,若算法的执行时间是常数级,不依赖数据量的大小,则时间复杂度为 $O(1)$;若算法的执行时间与数据量为线性关系,则时间复杂度为 $O(n)$;对数级、平方级、立方级、指数级的时间复杂度分别为 $O(\log_2 n)$、$O(n^2)$、$O(n^3)$、$O(2^n)$。这些函数按数量级递增排列具有下列关系:

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

循环语句的时间代价一般可用以下 3 条原则进行分析。

(1) 一个循环的时间代价=循环次数每次执行的基本指令数目。

(2) 多个并列的循环的时间代价=每个循环的时间代价之和。

(3) 多层嵌套循环的时间代价=每层循环的时间代价之积。

2. 算法的空间复杂度

算法的空间复杂度(Space Complexity)是指算法执行时所占用的额外存储空间量随问题规模的变化而变化的趋势。

执行一个算法所需要的存储空间主要包含以下两个部分。

(1) 固定空间部分：主要包括算法的程序指令、常量、变量所占的空间，与所处理问题的规模无关。

(2) 可变空间部分：主要包括输入的数据元素占用的空间和程序运行过程中额外的存储空间，与处理问题的规模有关。

在计算算法的空间复杂度时只考虑与算法相关的存储空间部分。算法本身占用的空间与实现算法的语言和描述语句有关，输入的数据元素与所处理的问题的规模有关，它们都不会随算法的改变而改变，所以将程序运行过程中额外的存储空间作为算法空间复杂度的量度。

当问题的规模为 n 时，$S(n)$表示此时算法占用的存储空间量，称为算法的空间复杂度，当 n 增大时 $S(n)$也随之增大。

通常采用算法的渐进分析中的大写的 O 表示法作为算法空间复杂度的渐进度量值，称为算法的渐进空间复杂度，记作 $S(n)=O(f(n))$，其分析与算法的渐进时间复杂度相同。

【例 1.4】 设 n 为正整数，试确定下列程序段中语句“$x+=1$”的频度。

(1)

```
for i in range(n):
    for j in range(i):
        x += 1
```

(2)

```
for i in range(n):
    for j in range(i):
        for k in range(n):
            x += 1
```

(3)

```
i,j = 0,1
while i + j <= n:
    x += 1
    if i > j:
        j += 1
    else:
        i += 1
```

(4)

```
x,y = 91,100
while y > 0:
    if(x > 100):
        x -= 10
        y -= 1
    else:
        x += 1
```

解：

(1) i 为 1 时，j 值只能取 1，语句执行一次；i 为 2 时，j 可取 1 或 2，语句执行两次；i 为

n 时,j 可取 1、2、…,n,语句执行 n 次,所以语句频度$=1+2+\cdots+n=n(n+1)/2$。

(2) i 为 1 时,j 值只能取 1,k 值可取 1、2、…、n,语句执行 n 次;i 为 2 时,j 可取 1 或 2,k 值可取 1、2、…、n,语句执行 $2n$ 次;i 为 n 时,j 可取 1、2、…、n,k 值可取 1、2、…、n 语句执行 $n\times n$ 次,所以语句频度$=n^2(n+1)/2$。

(3) i 与 j 初始和为 1,其后每循环一次 i 和 j 中有且仅有一个值增 1,即 i 与 j 的和增 1。由于循环条件为 $i+j\leqslant n$,因此循环共执行 n 次。所以,语句频度为 n。

(4) 分析 y 的初始值为 100,当 $y\leqslant 0$ 时循环结束,“x+=1”每执行 10 次 y 减小 1,所以“x+=1”的语句频度为 1000。

小　　结

(1) 数据结构包括逻辑结构和存储结构两个方面。数据的逻辑结构分为集合、线性结构、树结构和图形结构 4 种。数据的存储结构分为顺序存储结构、链式存储结构、索引存储结构和散列存储结构 4 种。

(2) 集合中的数据元素是相互独立的,在线性结构中数据元素具有“一对一”的关系,在树结构中数据元素具有“一对多”的关系,在图形结构中数据元素具有“多对多”的关系。

(3) 抽象数据类型描述了数据的特性和数据操作的语法规则,隐藏了数据的存储结构和操作的实现细节。在 Python 语言中抽象数据类型用抽象类的定义来实现。

(4) 算法的设计应该满足正确性、健壮性、高效率和可读性 4 个目标。

(5) 算法分析主要包括对时间复杂度和空间复杂度的分析,通常用大写的 O 表示法进行表示。

习　题　1

一、选择题

1. (　　)是数据的基本单位。

A. 数据元素　　B. 数据对象　　C. 数据项　　D. 数据结构

2. (　　)是数据的不可分割的最小单位。

A. 数据元素　　B. 数据对象　　C. 数据项　　D. 数据结构

3. 若采用非顺序映象,则数据元素在内存中占用的存储空间(　　)。

A. 一定连续　　B. 一定不连续　　C. 可连续可不连续

4. 若采用顺序映象,则数据元素在内存中占用的存储空间(　　)。

A. 一定连续　　B. 一定不连续　　C. 可连续可不连续

5. 在数据结构中从逻辑上可以把数据结构分为(　　)。

A. 动态结构和静态结构　　B. 紧凑结构和非紧凑结构

C. 线性结构和非线性结构　　D. 内部结构和外部结构

6. 在树结构中数据元素间存在(　　)的关系。

A. 一对一　　B. 一对多

C. 多对多　　D. 除同属一个集合外别无关系

7. 下列说法中错误的是(　　)。

A. 数据对象是数据的子集

B. 数据元素间的关系在计算机中的映像即为数据的存储结构

C. 非顺序映像的特点是借助指示元素存储地址的指针来表示数据元素间的逻辑关系

D. 抽象数据类型指一个数学模型及定义在该模型上的一组操作

8. 计算机算法指的是(　　)。

A. 计算方法　　B. 排序方法

C. 解决问题的有限运算序列　　D. 调度方法

9. 下列不属于算法特性的是(　　)。

A. 有穷性　　B. 确定性　　C. 零或多个输入　　D. 健壮性

10. 算法分析的目的是(　　)。

A. 找出数据结构的合理性　　B. 研究算法中的输入和输出的关系

C. 分析算法的效率以求改进　　D. 分析算法的易读性和文档性

11. 算法分析的两个主要方面是(　　)。

A. 空间复杂度和时间复杂度　　B. 正确性和简明性

C. 可读性和文档性　　D. 数据复杂性和程序复杂性

12. 算法的计算量的大小称为算法的(　　)。

A. 效率　　B. 复杂性　　C. 现实性　　D. 难度

13. 在下面的程序段中,对 x 的赋值语句的频度为(　　)。

```
for i in range(n):
    for j in range(n)
        x = x + 1
```

A. 2^n　　B. n　　C. n^2　　D. $\log n$

14. 设 n 为正整数,则以下程序段中最后一行的语句频度在最坏情况下是(　　)。

```
for i in range(n-1,0,-1):
    for j in range(1,i+1):
        if A[j] > A[j+1]:
            A[j] ←→ A[j+1]
```

A. n　　B. $n(n-1)/2$　　C. $n(n+1)/2$　　D. n^2

二、填空题

1. 数据逻辑结构包括________、________、________和________4 种类型,树形和图形结构合称________。

2. 对于给定的 n 个元素,可以构造出的逻辑结构有________、________、________和________4 种。

3. 算法的 5 个重要特性是________、________、________、________和________。

4. 评价算法的性能从利用计算机资源角度看主要从________方面进行分析。

5. 线性结构中元素之间存在________关系,树结构中元素之间存在________关系,图形结构中元素之间存在________关系。

6. 所谓数据的逻辑结构指的是数据元素之间的________。

7. 在线性结构中,开始节点________直接前驱节点,其余每个节点有且只有________个直接前驱节点。

8. 在树结构中,根节点只有________,根节点无前驱,其余每个节点有且只有________直接前驱节点;叶子节点没有________节点,其余每个节点的后继节点可以________。

9. 在图形结构中,每个节点的前驱节点和后继节点可以有________。

10. 存储结构是逻辑结构的________实现。

11. 一个算法的时空性能是指该算法的________和________。

12. 在一般情况下一个算法的时间复杂性是________的函数。

三、算法设计题

1. 判断 n 是否为一个素数,若是则返回逻辑值 true,否则返回逻辑值 false,写出算法并计算算法的时间复杂度。

2. 设计一个算法,计算 $\sum_{i=1}^{n} i!$ 的值,并计算算法的时间复杂度。

3. 设计一个算法,计算 $\sum_{i=1}^{n} i!$ 的值,并计算算法的空间复杂度。

4. 设计一个算法,求出满足不等式 $1+2+3+\cdots+i \geqslant n$ 的最小 i 值,并计算算法的时间复杂度。

5. 设计一个算法,打印出一个具有 n 行的乘法表,第 i 行($1 \leqslant i \leqslant n$)中有 $n-i+1$ 个乘法项,每个乘法项为 i 与 j($i \leqslant j \leqslant n$)的乘积,并计算算法的时间复杂度。

第2章 线 性 表

2.1 线性表及其基本操作

2.1.1 线性表的基本概念

线性表(Linear List)是其组成元素间具有线性关系的一种线性结构,是由 n 个具有相同数据类型的数据元素 a_0、a_1、…、a_{n-1} 构成的有限序列,一般表示为:

$$\{a_0, a_1, \cdots a_i, a_{i+1}, \cdots, a_{n-1}\}$$

其中,数据元素 a_i 可以是字母、整数、浮点数、对象或其他更复杂的信息,i 代表数据元素在线性表中的位序号($0 \leqslant i < n$),n 是线性表的元素个数,称为线性表的长度,当 $n=0$ 时线性表为空表。例如英文字母表{A,B,…,Z}是一个表长为 26 的线性表。又如表 2.1 所示的书籍信息表,这个信息表中的所有记录序列构成了一个线性表,线性表中的每个数据元素都是由书名、作者、出版社、价格 4 个数据项构成的记录。

表 2.1 书籍信息表

书名	作者	出版社	价格
软件工程实用教程	吕云翔	清华大学出版社	49.00

线性表中的数据元素具有线性的“一对一”的逻辑关系,是与位置有关的,即第 i 个元素 a_i 处于第 $i-1$ 个元素 a_{i-1} 的后面和第 $i+1$ 个元素 a_{i+1} 的前面。这种位置上的有序性就是一种线性关系,可以用二元组表示为 $L=(D,R)$,其中有以下关系:

$$D = \{a_i \mid 0 \leqslant i < n\}$$
$$R = \{r\}$$
$$r = \{< a_i, a_{i-1} > \mid 0 \leqslant i < n-1\}$$

对应的逻辑结构如图 2.1 所示。

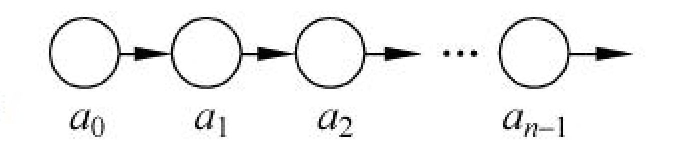


图 2.1 线性表的逻辑结构图

在线性表 $\{a_0, a_1, \cdots, a_{n-1}\}$ 中,a_0 为开始节点,没有前驱元素,a_{n-1} 为终端节点,没有后继元素。除开始节点和终端节点外,每个数据元素 a_i 都有且仅有一个前驱元素和后继元素。

2.1.2 抽象数据类型描述

线性表的抽象数据 Python 描述如下:

```
from abc import ABCMeta,abstractmethod,abstractproperty
```

```
class IList(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        pass
    @abstractmethod
    def isEmpty(self):
        pass
    @abstractmethod
    def length(self):
        pass
    @abstractmethod
    def get(self,i):
        pass
    @abstractmethod
    def insert(self,x):
        pass
    @abstractmethod
    def remove(self,i):
        pass
    @abstractmethod
    def indexOf(self,x):
        pass
    @abstractmethod
    def display(self):
        pass
```

【例 2.1】 有线性表 $A=\{1,2,3,4,5,6,7\}$,求 length()、isEmpty()、get(3)、indexOf(4)、display()、insert(2,7)和 remove(4)等基本运算的执行结果。

解:

```
length() = 7;
isEmpty()返回 false;
get(3)返回 4;
indexOf(4)返回 3;
display()输出 1,2,3,4,5,6,7;
insert(2,7)执行后线性表 A 变为 1,2,7,3,4,5,6,7;
remove(4)执行后线性表 A 变为 1,2,3,4,6,7。
```

2.1.3 线性表的存储和实现

在 2.1.2 节中,线性表的抽象数据 Python 抽象类包含了线性表的主要基本操作,如果要使用这个接口,还需要具体的类来实现。线性表的 Python 抽象类的实现方法主要有以下两种。

(1) 基于顺序存储的实现。

(2) 基于链式存储的实现。

2.2 线性表的顺序存储

2.2.1 顺序表

1. 定义

线性表的顺序存储结构是把线性表中的所有元素按照其逻辑顺序依次存储到计算机的内存单元中指定存储位置开始的一块连续的存储空间中，称为顺序表。顺序表用一组连续的内存单元依次存放数据元素，元素在内存中的物理存储次序和它们在线性表中的逻辑次序一致，即元素 a_i 与其前驱元素 a_{i-1} 和后继元素 a_{i+1} 的存储位置相邻，如图 2.2 所示。

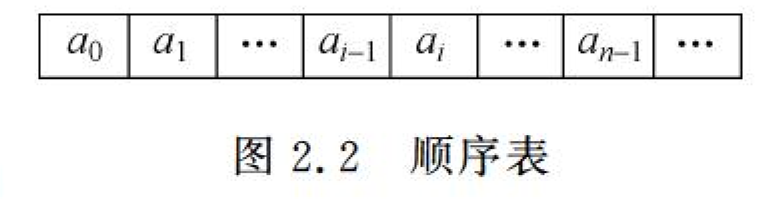


图 2.2 顺序表

又因为数据表中的所有数据元素具有相同的数据类型，所以只要知道顺序表的基地址和数据元素所占存储空间的大小即可计算出第 i 个数据元素的地址，可表示为：

$$\text{Loc}(a_i)=\text{Loc}(a_0)+i\times c,\text{其中 } 0\leqslant i\leqslant n-1$$

其中，$\text{Loc}(a_i)$是数据元素 a_i 的存储地址，$\text{Loc}(a_0)$是数据元素 a_0 的存储地址，即顺序表的基地址，i 为元素位置，c 为一个数据元素占用的存储单元。

可以看出，计算一个元素地址所需的时间为常量，与顺序表的长度 n 无关；存储地址是数据元素位序号 i 的线性函数。因此，存取任何一个数据元素的时间复杂度为 $O(1)$，顺序表是按照数据元素的位序号随机存取的结构。

2. 特点

(1) 在线性表中逻辑上相邻的元素在物理存储位置上也同样相邻。

(2) 可按照数据元素的位序号进行随机存取。

(3) 进行插入、删除操作需要移动大量的数据元素。

(4) 需要进行存储空间的预先分配，可能会造成空间浪费，但存储密度较高。

3. 描述

可以使用数组来描述线性表的顺序存储结构。在程序设计语言中，数组是一种构造数据类型。数组存储具有相同数据类型的元素集合，数组的存储单元个数称为数组长度，每个存储单元的地址是连续的，每个元素连续存储。数组通过下标识别元素，元素的下标是其存储单元序号，表示元素在数组中的位置。一维数组使用一个下标唯一确定一个元素，二维数组使用两个下标唯一确定一个元素。

下面是顺序表类的 Python 语言描述：

```
class SqList(IList):
    def __init__(self,maxsize):
        self.curLen = 0 # 顺序表的当前长度
        self.maxSize = maxsize # 顺序表的最大长度
        self.listItem = [None] * self.maxSize # 顺序表储存空间
    def clear(self):
        '''将线性表置成空表'''
        self.curLen = 0
```

```
    def isEmpty(self):
        '''判断线性表是否为空表'''
        return self.curLen == 0
    def length(self):
        '''返回线性表的长度'''
        return self.curLen
    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        if i < 0 or i > self.curLen - 1:
            raise Exception("第 i 个元素不存在")
        return self.listItem[i]
    def insert(self,i,x):
        '''插入 x 作为第 i 个元素'''
        if self.curLen == self.maxSize:
            raise Exception("顺序表满")
        if i < 0 or i > self.curLen:
            raise Exception("插入位置非法")
        for j in range(self.curLen,i - 1, - 1):
            self.listItem[j] = self.listItem[j - 1]
        self.listItem[i] = x
        self.curLen += 1
    def remove(self,i):
        '''删除第 i 个元素'''
        if i < 0 or i > self.curLen - 1:
            raise Exception("删除位置非法")
        for j in range(i,self.curLen):
            self.listItem[j] = self.listItem[j + 1]
        self.curLen -= 1
    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        for i in range(self.curLen):
            if self.listItem[i] == x:
                return i
        return - 1
    def display(self):
        '''输出线性表中各个数据元素的值'''
        for i in range(self.curLen):
            print(self.listItem[i],end = ' ')
```

2.2.2 顺序表的基本操作实现

1. 插入操作

插入操作 insert(i,x)是在长度为 n 的顺序表的第 i 个数据元素之前插入值为 x 的数据元素,其中 $0 \leqslant i \leqslant n$,当 $i=0$ 时在表头插入,当 $i=n$ 时在表尾插入,在插入操作完成后表长加 1,顺序表的逻辑结构由 $\{a_0, a_1, \cdots, a_{i-1}, a_i, \cdots, a_{n-1}\}$ 变成了 $\{a_0, a_1, \cdots, a_{i-1}, x, a_i, \cdots, a_{n-1}\}$,如图 2.3 所示。

其主要步骤如下。

(1) 判断顺序表的存储空间是否已满,若已满则抛出异常。

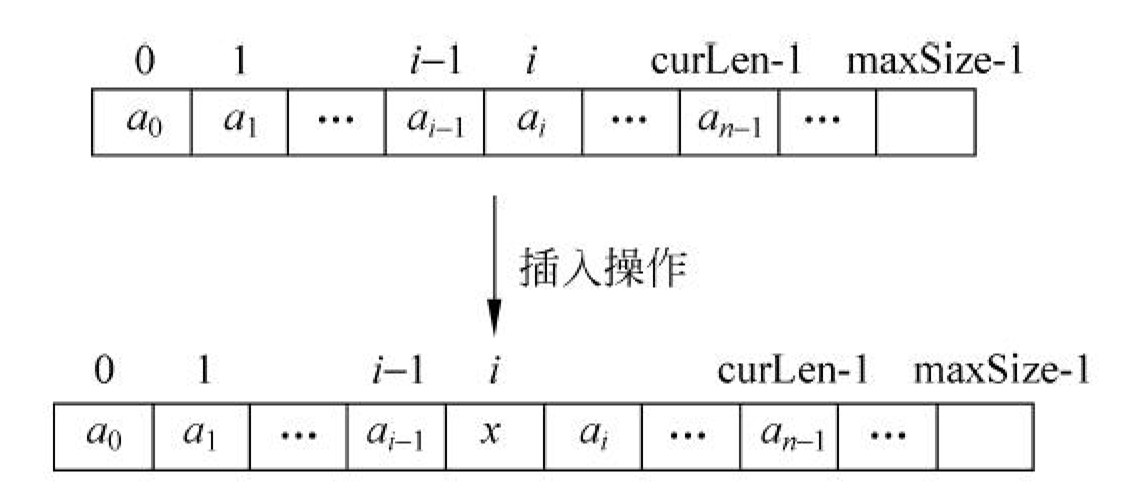


图 2.3 插入操作前后的顺序表存储结构图

(2) 判断参数 i 的值是否满足 $0 \leqslant i \leqslant \mathrm{curLen}$，若不满足则抛出异常。

(3) 将插入位置及其之后的所有数据元素后移一个存储位置。

(4) 在位置 i 处插入新的数据元素 x。

(5) 表长加 1。

【算法 2.1】 顺序表的插入操作算法。

```
1  def insert(self,i,x):
2      '''插入 x 作为第 i 个元素'''
3      if self.curLen == self.maxSize: # 判断顺序表的存储空间是否已满
4          raise Exception("顺序表满")
5      if i < 0 or i > self.curLen: # 判断参数的值是否满足
6          raise Exception("插入位置非法")
7      for j in range(self.curLen,i - 1, - 1):
8          self.listItem[j] = self.listItem[j - 1] # 将插入位置及之后的数据元素后移一个存储位置
9      self.listItem[i] = x # 在位置处插入新的数据元素
10     self.curLen += 1 # 表长加 1
```

算法的时间复杂度分析如下。

分析算法可以看出，在对顺序表进行插入操作时，时间花费主要用于第 8 行的数据元素的移动上。假设顺序表的表长为 n，若插入在表头，则需要移动 n 个元素，若插入在表尾，则需要移动 0 个元素。设插入位置为 i，则第 8 行语句的执行次数为 $n-i$。所以每次插入操作数据元素的平均移动次数为：

$$\sum_{i=0}^{n} p_i(n-i)$$

其中，p_i是在顺序表的第 i 个存储位置插入数据元素的概率，假设每个插入位置出现的概率相同，即为$\frac{1}{n+1}$，可得：

$$\frac{1}{n+1}\sum_{i=0}^{n}(n-i)=\frac{n}{2}$$

即在等概率情况下，插入一个数据元素平均需要移动顺序表数据元素的一半，算法的时间复杂度为 $O(n)$。

【例 2.2】 设 A 是一个线性表$\{a_0, a_1, \cdots, a_n\}$，采用顺序存储结构，则在等概率的前提下平均每插入一个元素需要移动的元素个数为多少？若元素插在 a_i 与 a_{i+1} 之间($1 \leqslant i \leqslant n$)的概率为$\frac{n-i}{\frac{n(n+1)}{2}}$，则平均每插入一个元素所要移动的元素个数又是多少？

解：

分析可得：

$$\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$$

$$\sum_{i=1}^{n}\frac{(n-i)^2}{n(n+1)/2}=(2n+1)/3$$

2. 删除操作

删除操作 remove(i,x)是将长度为 n 的顺序表的第 i 个数据元素删除，其中 $0\leqslant i\leqslant n-1$，删除操作完成后表长减 1，顺序表的逻辑结构由$\{a_0,a_1,\cdots,a_{i-1},a_i,\cdots,a_{n-1}\}$变成了$\{a_0,a_1,\cdots,a_{i-1},a_{i+1},\cdots,a_{n-1}\}$，如图 2.4 所示。

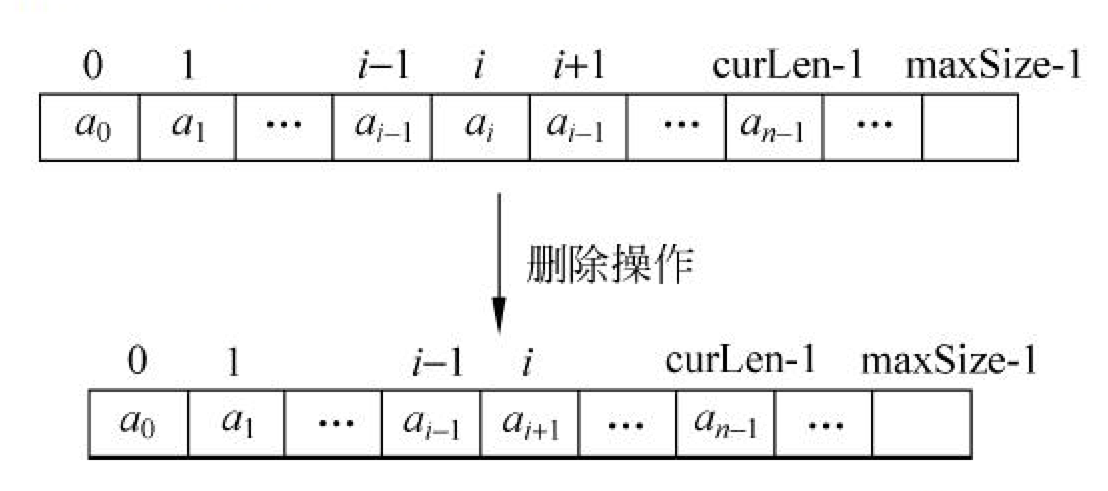


图 2.4 删除操作前后的顺序表存储结构图

其主要步骤如下。

(1) 判断参数 i 是否满足 $0\leqslant i\leqslant$ curLen-1，若不满足则抛出异常。

(2) 将第 i 个数据元素之后的数据元素都向前移动一个存储单元。

(3) 表长减 1。

【算法 2.2】 顺序表的删除操作算法。

```
1  def remove(self,i):
2      '''删除第 i 个元素'''
3      if i < 0 or i > self.curLen - 1:
4          raise Exception("删除位置非法")
5      for j in range(i,self.curLen):
6          self.listItem[j] = self.listItem[j + 1]
7      self.curLen -= 1
```

算法的时间复杂度分析如下。

分析算法可以看出，删除数据元素的时间花费主要用于第 6 行的数据元素的移动上。若顺序表的表长为 n，删除表头数据元素需要移动 n 个数据元素，删除表尾数据元素需要移动 0 个数据元素。设删除位置为 i，则第 6 行语句的执行次数为 $n-i-1$。所以每次删除操作数据元素的平均移动次数为：

$$\sum_{i=0}^{n-1}p_i(n-i-1)$$

其中，p_i是删除顺序表的第 i 个存储位置的数据元素的概率，假设每个删除位置出现的概率相同，即为$\frac{1}{n}$，可得：

$$\frac{1}{n}\sum_{i=0}^{n-1}(n-i-1)=\frac{n-1}{2}$$

即在等概率情况下，删除一个数据元素平均需要移动的顺序表的数据元素为$\frac{n-1}{2}$个，算法的时间复杂度为$O(n)$。

3. 查找操作

查找操作 indexOf(x)是在长度为n的顺序表中寻找初次出现的数据元素值为x的数据元素的位置。

其主要步骤为将x与顺序表中的每一个数据元素的值进行比较，若相等，则返回该数据元素的位置；若比较结束未找到等值的数据元素，返回-1。

【算法 2.3】 顺序表的查找操作算法。

```
def indexOf(self,x):
    '''返回元素 x 首次出现的位序号'''
    for i in range(self.curLen):
        if self.listItem[i] == x:
            return i
    return -1
```

算法的时间复杂度分析如下。

查找操作的比较次数取决于元素位置。分析算法可以看出，查找操作的时间花费主要集中在数据元素的比较上，设顺序表的数据元素个数为n，则比较次数最少为1、最多为n。假设各数据元素的查找概率相等，则数据元素的平均比较次数为：

$$\sum_{i=1}^{n}\frac{i}{n}=\frac{n+1}{2}$$

即在等概率情况下，查找一个数据元素平均需要比较的顺序表的数据元素为$\frac{n+1}{2}$个，算法的时间复杂度为$O(n)$。

【例 2.3】 建立一个由 a～z 的 26 个字母组成的字母顺序表，求每个字母的直接前驱和直接后继，编程实现。

```
L = SqList(26)
for i in range(26):
    L.insert(i,chr(ord('a') + i))

while True:
    i = input("请输入需要查询元素的位序号:\n")
    i = int(i)
    if i > 0 and i < 25:
        print("第%s个元素的直接前驱为:%s" % (i,L.get(i-1)))
        print("第%s个元素的直接后继为:%s" % (i,L.get(i+1)))
    elif i == 0:
        print("第%s个元素的直接前驱不存在" % (i,))
        print("第%s个元素的直接后继为:%s" % (i,L.get(i+1)))
    elif i == 25:
        print("第%s个元素的直接前驱为:%s" % (i,L.get(i-1)))
        print("第%s个元素的直接后继不存在" % (i,))
    else:
```

```
            print("位置非法")
```

【例 2.4】 建立一个顺序表,表中数据为 5 个学生的成绩{89,93,92,90,100},然后查找成绩为 90 的数据元素,并输出其在顺序表中的位置。

```
q = SqList(5)
for i,x in zip(range(5),[89,93,92,90,100]):
    q.insert(i,x)
res = q.indexOf(90)
if res ==- 1:
    print("顺序表中不存在成绩为 90 的数据元素")
else:
    print("顺序表中成绩为 90 的数据元素的位置为: %s" % res)
```

综上所述,顺序表具有较好的静态特性、较差的动态特性。

(1) 顺序表利用元素的物理存储次序反映线性表元素的逻辑关系,不需要额外的存储空间进行元素间关系的表达。顺序表是随机存储结构,存取元素 a_i 的时间复杂度为 $O(1)$,并且实现了线性表抽象数据类型所要求的基本操作。

(2) 插入和删除操作的效率很低,每插入或删除一个数据元素,元素的移动次数较多,平均移动顺序表中数据元素个数的一半;并且数组容量不可更改,存在因容量小造成数据溢出或者因容量过大造成内存资源浪费的问题。

2.3 线性表的链式存储和实现

采用链式存储方式存储的线性表称为链表,链表用若干地址分散的存储单元存储数据元素,逻辑上相邻的数据元素在物理位置上不一定相邻,必须采用附加信息表示数据元素之间的逻辑关系,因此链表的每一个节点不仅包含元素本身的信息,即数据域,而且包含元素之间逻辑关系的信息,即逻辑上相邻节点地址的指针域。

2.3.1 单链表

单链表是指节点中只包含一个指针域的链表,指针域中储存着指向后继节点的指针。单链表的头指针是线性表的起始地址,是线性表中第一个数据元素的存储地址,可作为单链表的唯一标识。单链表的尾节点没有后继节点,所以其指针域值为 None。

为了使操作简便,在第一个节点之前增加头节点,单链表的头指针指向头节点,头节点的数据域不存放任何数据,指针域存放指向第一个节点的指针。空单链表的头指针 head 为 None。图 2.5 为不带头节点的单链表的存储示意图,图 2.6 为带头节点的单链表的存储示意图。

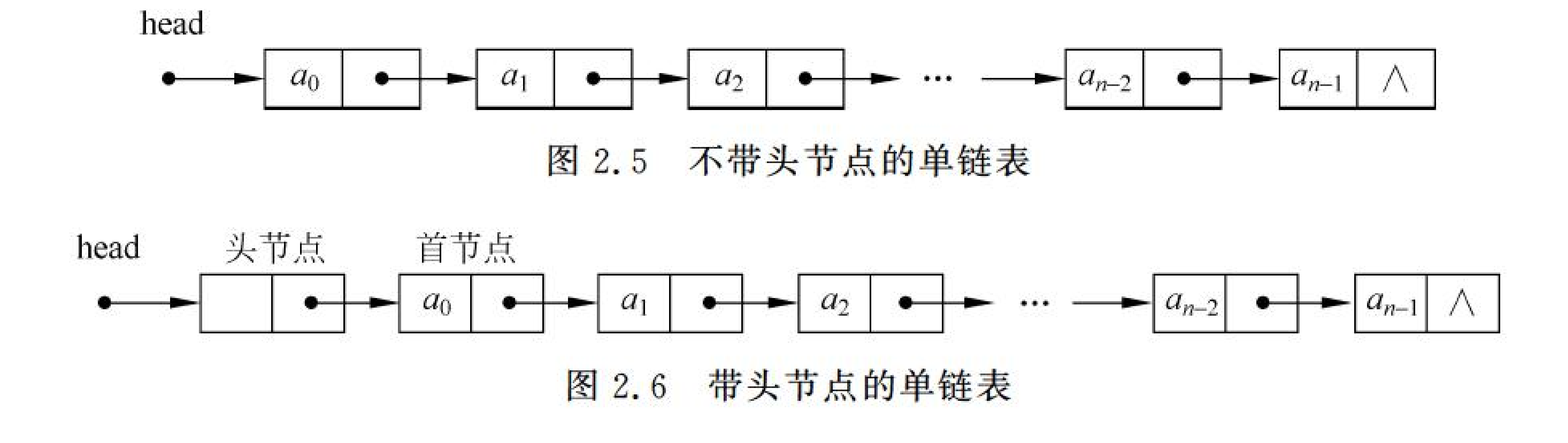


图 2.5　不带头节点的单链表

图 2.6　带头节点的单链表

单链表的节点的存储空间是在插入和删除过程中动态申请和释放的，不需要预先分配，从而避免了顺序表因存储空间不足需要扩充空间和复制元素的过程，避免了顺序表因容量过大造成内存资源浪费的问题，提高了运行效率和存储空间的利用率。

1. 节点类描述

```
class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next
```

2. 单链表类描述

```
class LinkList(IList):
    def __init__(self):
        self.head = Node() # 构造函数初始化头节点

    def create(self,l,order):
        if order:
            self.create_tail(l)
        else:
            self.create_head(l)

    def create_tail(self,l):
        pass

    def create_head(self,l):
        pass

    def clear(self):
        '''将线性表置成空表'''
        self.head.data = None
        self.head.next = None

    def isEmpty(self):
        '''判断线性表是否为空表'''
        return self.head.next == None

    def length(self):
        '''返回线性表的长度'''
        p = self.head.next
        length = 0
        while p is not None:
            p = p.next
            length += 1
        return length

    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        pass

```

```
    def insert(self,i,x):
        '''插入 x 作为第 i 个元素'''
        pass

    def remove(self,i):
        '''删除第 i 个元素'''
        pass

    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        pass

    def display(self):
        '''输出线性表中各个数据元素的值'''
        p = self.head.next
        while p is not None:
            print(p.data,end = ' ')
            p = p.next
```

2.3.2 单链表的基本操作实现

1. 查找操作

(1) 位序查找 get(i)是返回长度为 n 的单链表中第 i 个节点的数据域的值,其中 $0 \leqslant i \leqslant n-1$。由于单链表的存储空间不连续,因此必须从头节点开始沿着后继节点依次进行查找。

【算法 2.4】 位序查找算法。

```
def get(self,i):
    '''读取并返回线性表中的第 i 个数据元素'''
    p = self.head.next # p 指向单链表的首节点
    j = 0
    # 从首节点开始向后查找,直到 p 指向第 i 个节点或者 p 为 None
    while j < i and p is not None:
        p = p.next
        j += 1
    if j > i or p is None: # i 不合法时抛出异常
        raise Exception("第" + i + "个数据元素不存在")
    return p.data
```

(2) 查找操作 indexOf(x)是在长度为 n 的单链表中寻找初次出现的数据域值为 x 的数据元素的位置。

其主要步骤为将 x 与单链表中的每一个数据元素的数据域进行比较,若相等,则返回该数据元素在单链表中的位置;若比较结束未找到等值的数据元素,返回−1。

【算法 2.5】 按值查找。

```
def indexOf(self,x):
    '''返回元素 x 首次出现的位序号'''
    p = self.head.next
```

```
        j = 0
        while p is not None and not (p.data == x):
            p = p.next
            j += 1
        if p is not None:
            return j
        else:
            return -1
```

2. 插入操作

插入操作 insert(i,x)是在长度为 n 的单链表的第 i 个节点之前插入数据域值为 x 的新节点，其中 $0 \leqslant i \leqslant n$，当 $i=0$ 时，在表头插入，当 $i=n$ 时在表尾插入。

与顺序表相比，单链表不需要移动一批数据元素，而只需要改变节点的指针域，改变有序对，即可实现数据元素的插入，即 $<a_{i-1}, a_i>$ 转变为 $<a_{i-1}, x>$ 和 $<x, a_i>$，如图 2.7 所示。

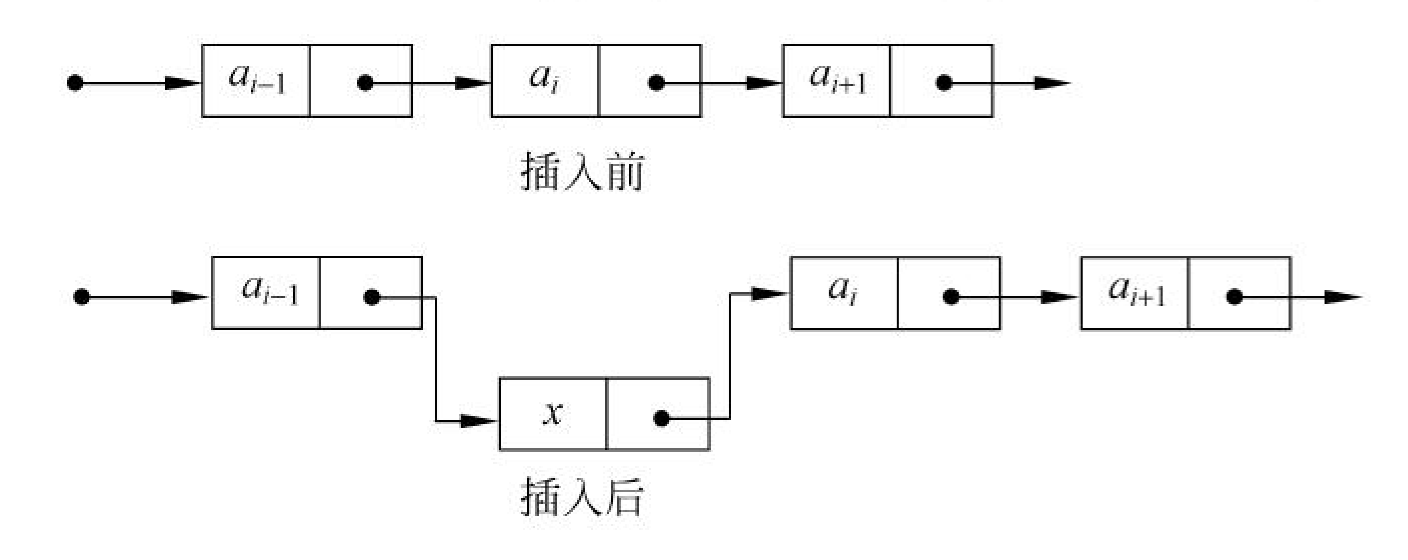


图 2.7　单链表上的插入

插入操作的主要步骤如下。

(1) 查找到插入位置的前驱节点，即第 $i-1$ 个节点。

(2) 创建数据域值为 x 的新节点。

(3) 修改前驱节点的指针域为指向新节点的指针，新节点的指针域为指向原第 i 个节点的指针。

【算法 2.6】　带头节点的单链表的插入操作。

```
def insert(self,i,x):
    '''插入 x 作为第 i 个元素'''
    p = self.head
    j = -1
    while p is not None and j < i - 1:
        p = p.next
        j += 1
    if j > i - 1 or p is None:
        raise Exception("插入位置不合法")
    s = Node(x,p.next)
    p.next = s
```

【算法 2.7】　不带头节点的单链表的插入操作。

```
def insert(self,i,x):
    p = self.head
    j = -1
    while p is not None and j < i - 1:
```

```
            p = p.next
            j += 1
    if j > i - 1 or p is None:
        raise Exception("插入位置不合法")
    s = Node(data = x)
    if i == 0:
        s.next = self.head
    else:
        s.next = p.next
        p.next = s
```

分析以上代码可以发现,由于链式存储采用的是动态存储分配空间,所以在进行插入操作之前不需要判断存储空间是否已满。

在带头节点的单链表上进行插入操作时,无论插入位置是表头、表尾还是表中,操作语句都是一致的;但是在不带头节点的单链表上进行插入操作时,在表头插入和在其他位置插入新节点的语句是不同的,需要分两种情况进行处理。

3. 删除操作

删除操作 remove(i,x)是将长度为 n 的单链表的第 i 个节点删除,其中 $0 \leqslant i \leqslant n-1$。

与顺序表相比,单链表不需要移动一批数据元素,而只需要改变节点的指针域,实现有序对的改变,即可删除节点,即$<a_{i-1},a_i>$和$<a_i,a_{i+1}>$转变为$<a_{i-1},a_{i+1}>$,如图 2.8 所示。

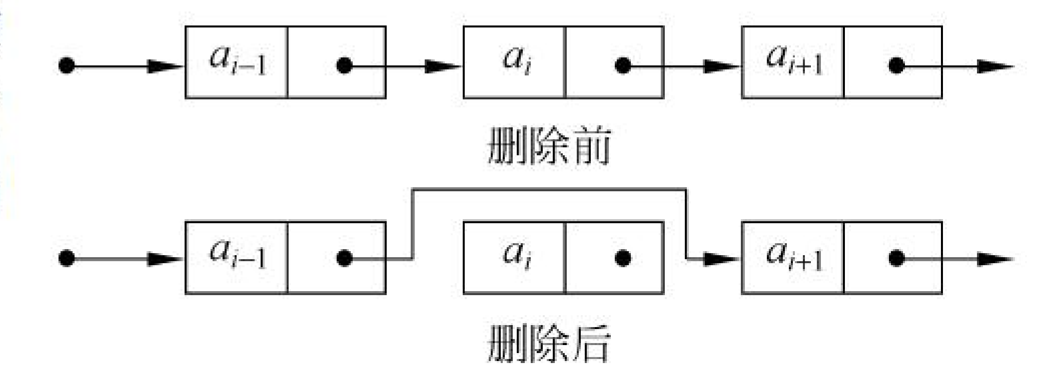


图 2.8　单链表的删除操作

其主要步骤如下。

(1) 判断单链表是否为空。

(2) 查找待删除节点的前驱节点。

(3) 修改前驱节点的指针域为待删除节点的指针域。

【算法 2.8】 删除操作。

```
def remove(self,i):
    '''删除第 i 个元素'''
    p = self.head
    j = -1
    # 寻找第 i 个节点的前驱节点
    while p is not None and j < i - 1:
        p = p.next
        j += 1
    if j > i - 1 or p.next is None:
        raise Exception("删除位置不合法")
    p.next = p.next.next
```

4. 单链表的建立操作

1) 头插法

将新节点插入到单链表的表头,读入的数据顺序与节点顺序相反。

【算法 2.9】 头插法。

```
def create_head(self,l):
```

```python
    for item in l:
        self.insert(0,item)
```

2）尾插法

将新节点插入到单链表的表尾，读入的数据顺序与节点顺序相同。

【算法 2.10】 尾插法。

```python
def create_tail(self,l):
    for item in l:
        self.insert(self.length(),item)
```

【例 2.5】 编程实现将列表中的元素构建成一个有序的单链表。

```python
data = [i for i in range(10)]
ll = LinkList()
ll.create(data,True)
ll.display()
```

2.3.3 其他链表

1. 循环链表

循环链表与单链表的结构相似，只是将链表的首尾相连，即尾节点的指针域为指向头节点的指针，从而形成了一个环状的链表。

循环链表与单链表的操作算法基本一致，判定循环链表中的某个节点是否为尾节点的条件不是它的后继节点为空，而是它的后继节点是否为头节点。

在实现循环链表时可用头指针或尾指针或二者同时使用来标识循环链表，通常使用尾指针来进行标识，可简化某些操作。

2. 双向链表

双向链表的节点具有两个指针域，一个指针指向前驱节点，一个指针指向后继节点，使得查找某个节点的前驱节点不需要从表头开始顺着链表依次进行查找，减小时间复杂度。

1）节点类描述

```python
class DuLNode(object):
    def __init__(self,data = None,prior = None,next = None):
        self.data = data
        self.prior = prior
        self.next = next
```

2）双向链表的基本操作实现

其与单链表的不同之处主要在于进行插入和删除操作时每个节点需要修改两个指针域。

【算法 2.11】 插入操作。

```python
def insert(self,i,x):
    p = self.head
    j = -1
    # 寻找插入位置 i
```

```
    while p is not None and j < i:
        p = p.next
        j += 1
    if j > i or p is None:
        raise Exception("插入位置不合法")
    s = DuLNode(data = x)
    p.prior.next = s
    s.next = p
    p.prior = s
```

【算法 2.12】 删除操作。

```
def remove(self,i):
    p = self.head
    j = -1
    while p is not None and j < i:
        p = p.next
        j += 1
    if j > i or p is None:
        raise Exception("删除位置不合法")
    p.prior.next = p.next
    p.next.piror = p.piror
```

2.4 顺序表与链表的比较

顺序表与链表的比较如表 2.2 所示。

表 2.2 顺序表与链表的比较

	顺序表	链表
优点	(1) 可进行高效随机存取; (2) 存储密度高,空间开销小; (3) 实现简单,便于使用	(1) 灵活,可进行存储空间的动态分配; (2) 插入、删除效率高
缺点	(1) 需要预先分配存储空间; (2) 不便于进行插入和删除操作	(1) 存储密度低; (2) 不可按照位序号随机存取

小结

(1) 线性表是其组成元素间具有线性关系的一种线性结构,其实现方式主要为基于顺序存储的实现和基于链式存储的实现。

(2) 线性表的顺序存储结构称为顺序表,可用数组实现,可对数据元素进行随机存取,时间复杂度为 $O(1)$,在插入或删除数据元素时时间复杂度为 $O(n)$。

(3) 线性表的链式存储结构称为链表,不能直接访问给定位置上的数据元素,必须从头节点开始沿着后继节点进行访问,时间复杂度为 $O(n)$。在插入或删除数据元素时不需要移动任何数据元素,只需要更改节点的指针域即可,时间复杂度为 $O(1)$。

(4) 循环链表将链表的首尾相连，即尾节点的指针域为指向头节点的指针，从而形成了一个环状的链表。

(5) 双向链表的节点具有两个指针域，一个指针指向前驱节点，一个指针指向后继节点，使得查找某个节点的前驱节点不需要从表头开始顺着链表依次进行查找，减小时间复杂度。

习　题　2

一、选择题

1. 在一个长度为 n 的顺序存储的线性表中，向第 i 个元素($1\leqslant i\leqslant n+1$)位置插入一个新元素时需要从后向前依次后移(　　)个元素。

A. $n-i$　　B. $n-i+1$　　C. $n-i-1$　　D. i

2. 在一个长度为 n 的顺序存储的线性表中删除第 i 个元素($1\leqslant i\leqslant n$)时需要从前向后依次前移(　　)个元素。

A. $n-i$　　B. $n-i+1$　　C. $n-i-1$　　D. i

3. 在一个长度为 n 的线性表中顺序查找值为 x 的元素时，在等概率情况下查找成功时的平均查找长度(即需要比较的元素个数)为(　　)。

A. n　　B. $n/2$　　C. $(n+1)/2$　　D. $(n-1)/2$

4. 在一个长度为 n 的线性表中删除值为 x 的元素时需要比较元素和移动元素的总次数为(　　)。

A. $(n+1)/2$　　B. $n/2$　　C. n　　D. $n+1$

5. 在一个顺序表的表尾插入一个元素的时间复杂度为(　　)。

A. $O(n)$　　B. $O(1)$　　C. $O(n*n)$　　D. $O(\log_2 n)$

6. 若一个节点的引用为 p，它的前驱节点的引用为 q，则删除 p 的后继节点的操作为(　　)。

A. p=p.next.next　　B. p.next=p.next.next

C. q.next=p.next　　D. q.next=q.next.next

7. 假定一个多项式中 x 的最高次幂为 n，则在保存所有系数项的线性表表示中其线性表长度为(　　)。

A. $n+1$　　B. n　　C. $n-1$　　D. $n+2$

二、填空题

1. 对于当前长度为 n 的线性表，共包含有________个插入元素的位置，共包含有________个删除元素的位置。

2. 若经常需要对线性表进行表尾插入和删除运算，则最好采用________存储结构；若经常需要对线性表进行表头插入和删除运算，则最好采用________存储结构。

3. 由 n 个元素生成一个顺序表。若每次都调用插入算法把一个元素插入到表头，则整个算法的时间复杂度为________；若每次都调用插入算法把一个元素插入到表尾，则整个

算法的时间复杂度为________。

4. 由 n 个元素生成一个单链表。若每次都调用插入算法把一个元素插入到表头,则整个算法的时间复杂度为________;若每次都调用插入算法把一个元素插入到表尾,则整个算法的时间复杂度为________。

5. 对于一个长度为 n 的顺序存储的线性表,在表头插入元素的时间复杂度为________,在表尾插入元素的时间复杂度为________。

6. 对于一个单链接存储的线性表,在表头插入节点的时间复杂度为________,在表尾插入节点的时间复杂度为________。

7. 从一个顺序表和单链表中访问任一给定位置序号的元素(节点)的时间复杂度分别为________和________。

三、算法设计题

1. 修改从顺序存储的集合中删除元素的算法,要求在删除一个元素后检查数组空间的大小,若空间利用率小于40%并且数组长度大于 maxSize 时则释放数组的一半存储空间。

2. 编写顺序存储集合类 sequenceSet 中的构造方法,它包含有一维列表参数 a,该方法中给 setArray 数组分配的长度是 a 列表长度的1.5倍,并且根据 a 列表中所有不同的元素值建立一个集合。

3. 编写一个静态成员方法,返回一个顺序存储的集合 set 中所有元素的最大值,假定元素类型为 double。

4. 编写顺序存储集合类 sequenceSet 中的 copy 方法,它包含有一个参数 Set set,实现把 set 所指向的顺序集合的内容复制到当前集合中的功能。

5. 编写一个静态成员方法,实现两个顺序存储集合的差运算,并返回所求得的差集。

6. 编写一个静态成员方法,实现两个链式存储集合的差运算,并返回所求得的差集。

7. 编写一个带有主函数的程序,其中包含两个静态成员方法,分别为使用顺序和链接存储的线性表解决约瑟夫(Josephus)问题。约瑟夫问题:设有 n 个人围坐在一张圆桌周围,现从某个人开始从1报数,数到 m 的人出列(即离开座位,不参加以后的报数),接着从出列的下一个人开始重新从1报数,数到 m 的人又出列,如此下去直到所有人都出列为止,试求出这 n 个人的出列次序。

例如,当 $n=8$、$m=4$ 时,若从第一个人开始报数,假定 n 个人对应的编号依次为1、2、…、n,则得到的出列次序为“4,8,5,2,1,3,7,6”。

在每个解决约瑟夫问题的静态成员方法中,要求以整型对象 n、m 和 s 作为参数,n 表示开始参加报数的人数,m 为下一次要出列的人所报出的数字序号,s 为最开始报数的那个人的编号。

注意:人的座位是首尾相接的,所以报数是循环进行的,最后一个人报数后接着是最前面的一个人报数。

第3章 栈和队列

3.1 栈

3.1.1 栈的基本概念

栈是一种特殊的线性表，其插入、删除操作只能在表的尾部进行。在栈中允许进行插入、删除操作的一端称为栈顶，另一端称为栈底。在栈$\{a_0, a_1, \cdots, a_{n-1}\}$中$a_0$称为栈底元素，$a_{n-1}$称为栈顶元素。通常，栈的插入操作叫入栈，栈的删除操作叫出栈。

由于栈的插入和删除操作只允许在栈顶进行，每次入栈的元素即成为栈顶元素，每次最先出栈的总是栈顶元素，所以栈是一种后进先出的线性表。就像一摞盘子，每次将一个盘子摞在最上面，每次从最上面取一只盘子，不能从中间插进或者抽出。

3.1.2 栈的抽象数据类型描述

栈中的数据元素和数据间的逻辑关系与线性表相同，是由n个具有相同数据类型的数据元素构成的有限序列，栈的抽象数据类型的 Python 描述如下：

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IStack(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将栈置空'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断栈是否为空'''
        pass
    @abstractmethod
    def length(self):
        '''返回栈的数据元素个数'''
        pass
    @abstractmethod
    def peek(self):
        '''返回栈顶元素'''
        pass
    @abstractmethod
    def push(self,x):
```

```
        '''数据元素 x 入栈'''
        pass
    @abstractmethod
    def pop(self):
        '''将栈顶元素出栈并返回'''
        pass
    @abstractmethod
    def display(self):
        '''输出栈中的所有元素'''
        pass
```

栈的抽象数据 Python 抽象类包含了栈的主要基本操作,如果要使用这个类还需要具体的类来实现。栈的 Python 抽象类的实现方法主要有以下两种。

(1) 基于顺序存储的实现,为顺序栈。

(2) 基于链式存储的实现,为链栈。

3.1.3 顺序栈

1. 顺序栈类的描述

顺序栈用数组实现,因为入栈和出栈操作都是在栈顶进行,所以增加变量 top 来指示栈顶元素的位置,top 指向栈顶元素存储位置的下一个存储单元的位置,空栈时 top=0。

顺序栈类的 Python 语言描述如下:

```
class SqStack(IStack):
    def __init__(self,maxSize):
        self.maxSize = maxSize # 栈的最大存储单元个数
        self.stackElem = [None] * self.maxSize # 顺序栈存储空间
        self.top = 0 # 指向栈顶元素的下一个存储单元位置

    def clear(self):
        '''将栈置空'''
        self.top = 0

    def isEmpty(self):
        '''判断栈是否为空'''
        return self.top == 0

    def length(self):
        '''返回栈的数据元素个数'''
        return self.top

    def peek(self):
        '''返回栈顶元素'''
        if not self.isEmpty():
            return self.stackElem[self.top - 1]
        else:
            return None
    def push(self,x):
        '''数据元素 x 入栈'''
```

```
            pass

    def pop(self):
        '''将栈顶元素出栈并返回'''
        pass

    def display(self):
        '''输出栈中的所有元素'''
        for i in range(self.top - 1, - 1, - 1):
            print(self.stackElem[i],end = ' ')
```

2. 顺序栈基本操作的实现

1）入栈操作

入栈操作 push(x)是将数据元素 x 作为栈顶元素插入顺序栈中，主要操作如下。

(1) 判断顺序栈是否为满，若满则抛出异常。

(2) 将 x 存入 top 所指的存储单元位置。

(3) top 加 1。

图 3.1 显示了进行入栈操作时栈的状态变化。

【算法 3.1】 入栈操作。

```
def push(self,x):
    '''数据元素 x 入栈'''
    if self.top == self.maxSize:
        raise Exception("栈已满")
    self.stackElem[self.top] = x
    self.top += 1
```

2）出栈操作

出栈操作 pop()是将栈顶元素从栈中删除并返回，主要步骤如下。

(1) 判断顺序栈是否为空，若空则返回 null。

(2) top 减 1。

(3) 返回 top 所指的栈顶元素的值。

图 3.2 显示了进行出栈操作时栈的状态变化。

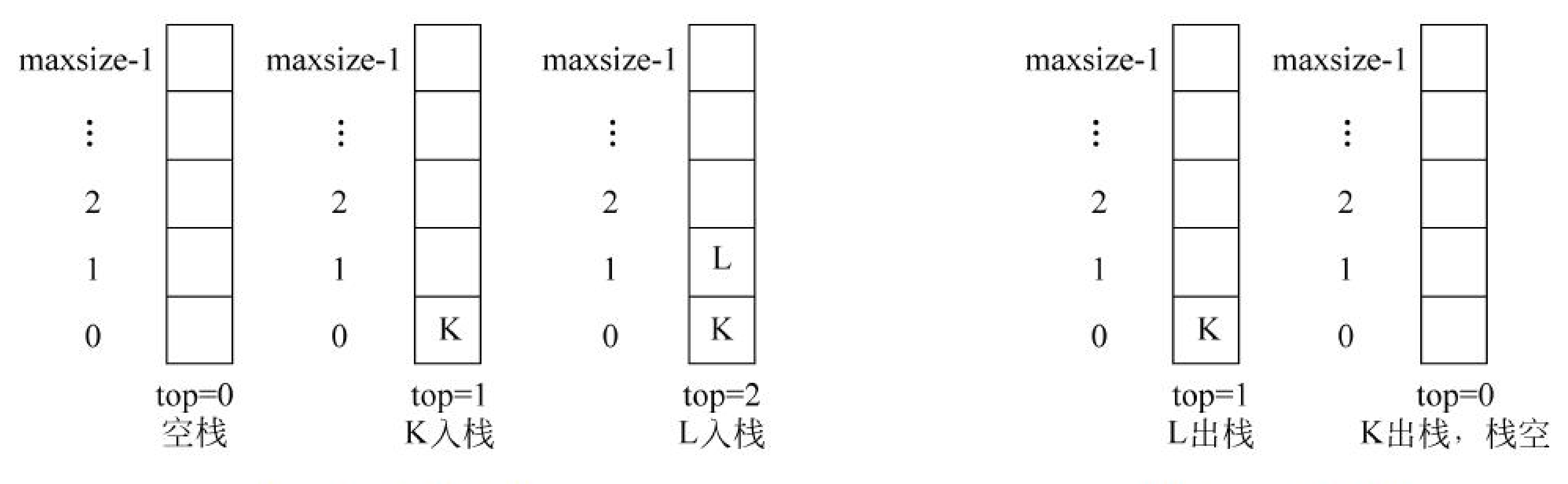


图 3.1 入栈操作　　图 3.2 出栈操作

【算法 3.2】 出栈操作。

```
def pop(self):
```

```
        '''将栈顶元素出栈并返回'''
        if self.isEmpty():
            return None
        self.top -= 1
        return self.stackElem[self.top]
```

分析可得,入栈和出栈操作的实现为顺序表的尾插入和尾删除,时间复杂度为 $O(1)$。

【例 3.1】 利用顺序栈实现括号匹配的语法检查。

解: 括号匹配是指程序中出现的括号,左、右括号的个数是相同的,并且需要先左后右依次出现。括号是可以嵌套的,一个右括号与其前面最近的一个左括号匹配,使用栈保存多个嵌套的左括号。

```
def isMatched(str):
    s = SqStack(100)
    for c in str:
        if c == '(':
            s.push('(')
        elif c == ')' and not s.isEmpty():
            s.pop()
        elif c == ')' and s.isEmpty():
            print("括号不匹配")
            return False
    if s.isEmpty():
        print("括号匹配")
        return True
    else:
        print("括号不匹配")
        return False
```

3.1.4 链栈

1. 链栈类的描述

采用链式存储结构的栈称为链栈,由于入栈和出栈只能在栈顶进行,不存在在栈的任意位置进行插入和删除的操作,所以不需要设置头节点,只需要将指针 top 指向栈顶元素节点,每个节点的指针域指向其后继节点即可。

链栈的存储结构如图 3.3 所示。

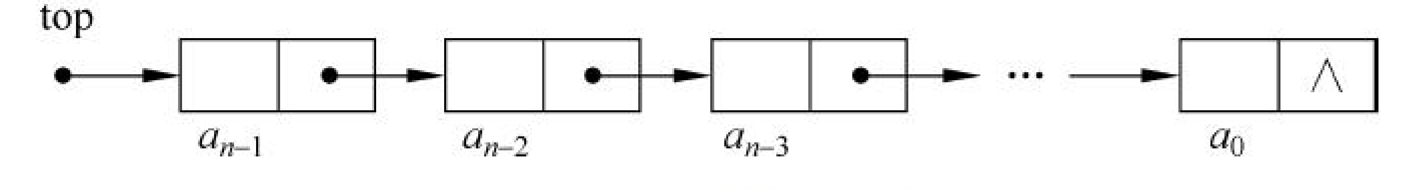


图 3.3 链栈的存储结构

实现 IStack 抽象类的链栈类的 Python 语言描述如下:

```
class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next

```

```python
class LinkStack(IStack):
    def __init__(self):
        self.top = None

    def clear(self):
        '''将栈置空'''
        self.top = None

    def isEmpty(self):
        '''判断栈是否为空'''
        return self.top is None

    def length(self):
        '''返回栈的数据元素个数'''
        i = 0
        p = self.top
        while p is not None:
            p = p.next
            i += 1
        return i

    def peek(self):
        '''返回栈顶元素'''
        return self.top

    def push(self,x):
        '''数据元素 x 入栈'''
        s = Node(x,self.top)
        self.top = s

    def pop(self):
        '''将栈顶元素出栈并返回'''
        if self.isEmpty():
            return None
        p = self.top
        self.top = self.top.next
        return p.data

    def display(self):
        '''输出栈中的所有元素'''
        p = self.top
        while p is not None:
            print(p.data,end = ' ')
            p = p.next
```

2. 链栈基本操作的实现

1）入栈操作

入栈操作 push(x)是将数据域值为 x 的节点插入到链栈的栈顶,主要步骤如下。

(1) 构造数据值域为 x 的新节点。

(2) 改变新节点和首节点的指针域,使新节点成为新的栈顶节点。

链栈进行入栈操作后的状态变化如图 3.4 所示。

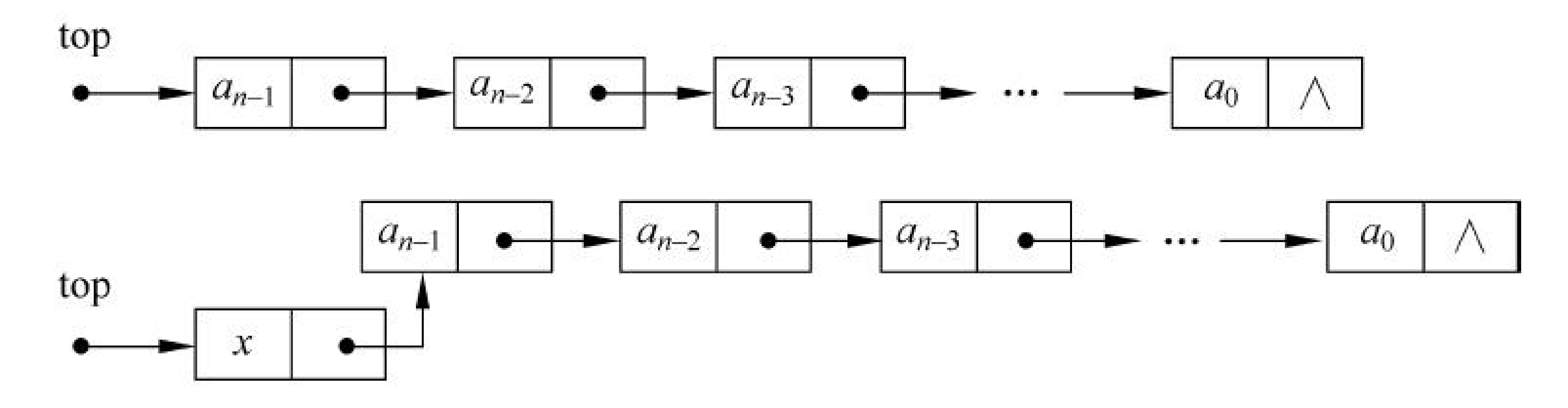


图 3.4 入栈操作

【算法 3.3】 入栈操作。

```
def push(self,x):
    '''数据元素 x 入栈'''
    s = Node(x,self.top)
    self.top = s
```

2) 出栈操作

出栈操作 pop()是将栈顶元素从链栈中删除并返回其数据域的值,主要步骤如下。

(1) 判断链栈是否为空,若空则返回 null。

(2) 修改 top 指针域的值,返回被删节点的数据域值。

链栈进行出栈操作后的状态变化如图 3.5 所示。

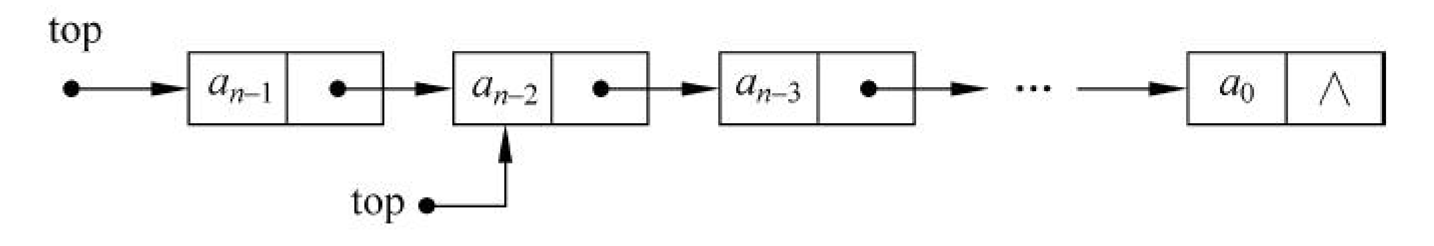


图 3.5 出栈操作

【算法 3.4】 出栈操作。

```
def pop(self):
    '''将栈顶元素出栈并返回'''
    if self.isEmpty():
        return None
    p = self.top
    self.top = self.top.next
    return p.data
```

分析可得,使用单链表实现栈,入栈和出栈操作的实现为单链表的头插入和头删除,时间复杂度为 $O(1)$。

【例 3.2】 设有编号为 1、2、3、4 的 4 辆列车顺序进入一个栈式结构的车站,具体写出这 4 辆列车开出车站的所有可能的顺序。

解:至少有 14 种。

全进之后再出的情况只有 1 种:4,3,2,1。

进 3 个之后再出的情况有 3 种:3,4,2,1; 3,2,4,1; 3,2,1,4。

进两个之后再出的情况有 5 种:2,4,3,1; 2,3,4,1; 2,1, 3,4; 2,1,4,3; 2,3,1,4。

进一个之后再出的情况有 5 种:1,4,3,2; 1,3,2,4; 1,3,4,2; 1,2,3,4; 1,2,4,3。

【例 3.3】 在执行操作序列 push(1)、push(2)、pop、push(5)、push(7)、pop、push(6)之后栈顶元素和栈底元素分别是什么?

解:操作序列的执行过程如图 3.6 所示。

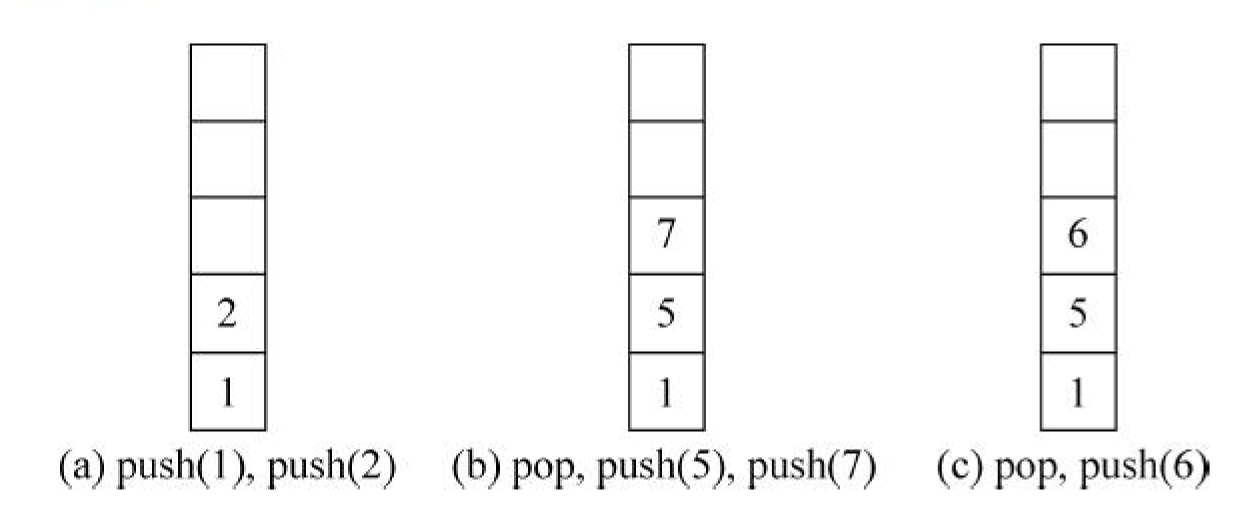


图 3.6 操作序列的执行过程

所以栈顶元素为 6,栈底元素为 1。

【例 3.4】 编程实现汉诺塔问题的求解。假设有 3 个分别命名为 x、y、z 的塔座,在塔座 x 上插有 n 个直径和序号均为 1、2、…、n 的圆盘。要求将塔座 x 上的 n 个圆盘借助塔座 y 移动到塔座 z 上,仍按照相同的序号排列,并且每次只能移动一个圆盘,在任何时候都不能将一个较大的圆盘压在较小的圆盘之上。

解:分析问题可知,当 $n=1$ 时只要将序号为 1 的圆盘从 x 直接移动到 z 即可;当 $n>1$ 时则需要将序号小于 n 的 $n-1$ 个圆盘移动到 y 上,再将序号为 n 的圆盘移动到 z 上,然后将 y 上的 $n-1$ 个圆盘移动到 z 上。如何将 $n-1$ 个圆盘移动到 z 上是一个和原问题相似的问题,只是规模变小,可以用同样的方法求解。代码如下:

```
def move(s,t):
    print("将",s,"塔座上最顶端圆盘移动到",t,"塔座上")

def hanoi(n,x,y,z):
    if n==1:
        move(x,z)
    else:
        hanoi(n-1,x,z,y) # 将 x 上 n-1 个圆盘从 x 移动到 y
        move(x,z) #将 1 个圆盘从塔座 x 移动到塔座 z
        hanoi(n-1,y,x,z) # 将 y 上 n-1 个圆盘从 y 移动到 z

hanoi(3,'x','y','z')
```

3.2 队　　列

3.2.1 队列的基本概念

队列是一种特殊的线性表,其插入操作只能在表的尾部进行,删除操作只能在表头进行。在队列中允许进行插入操作的一端称为队尾,允许进行删除操作的另一端称为队首。在队列$\{a_0,a_1,\cdots,a_{n-1}\}$中 a_0 称为队首元素,a_{n-1} 称为队尾元素。通常,队列的插入操作叫入队,队列的删除操作叫出队。没有数据元素的队列称为空队列。

由于插入和删除操作分别在队尾和队首进行,最先入队的元素总是最先出队,因此队列具有先进先出的特点。

3.2.2 队列的抽象数据类型描述

队列中的数据元素和数据间的逻辑关系与线性表相同,是由 n 个具有相同数据类型的数据元素构成的有限序列,队列的抽象数据类型的 Python 描述如下:

```python
from abc import ABCMeta,abstractmethod,abstractproperty

class IQueue(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将队列置空'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断队列是否为空'''
        pass
    @abstractmethod
    def length(self):
        '''返回队列的数据元素个数'''
        pass
    @abstractmethod
    def peek(self):
        '''返回队首元素'''
        pass
    @abstractmethod
    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        pass
    @abstractmethod
    def poll(self):
        '''将队首元素删除并返回其值'''
        pass
    @abstractmethod
    def display(self):
        '''输出队列中的所有元素'''
        pass
```

队列的抽象数据 Python 抽象类包含了队列的主要基本操作,如果要使用这个接口,还需要具体的类来实现。Python 抽象类的实现方法主要有以下两种。

(1) 基于顺序存储的实现,为顺序队列。

(2) 基于链式存储的实现,为链队列。

3.2.3 顺序队列

1. 顺序队列类的描述及实现

顺序队列的存储结构与顺序栈类似,可用数组实现,因为入队和出队操作分别在队尾和

队首进行,所以增加变量 front 来指示队首元素的位置,rear 指示队尾元素的下一个存储单元的位置。顺序队列进行入队操作的状态变化如图 3.7 所示,进行出队操作后的状态变化如图 3.8 所示。

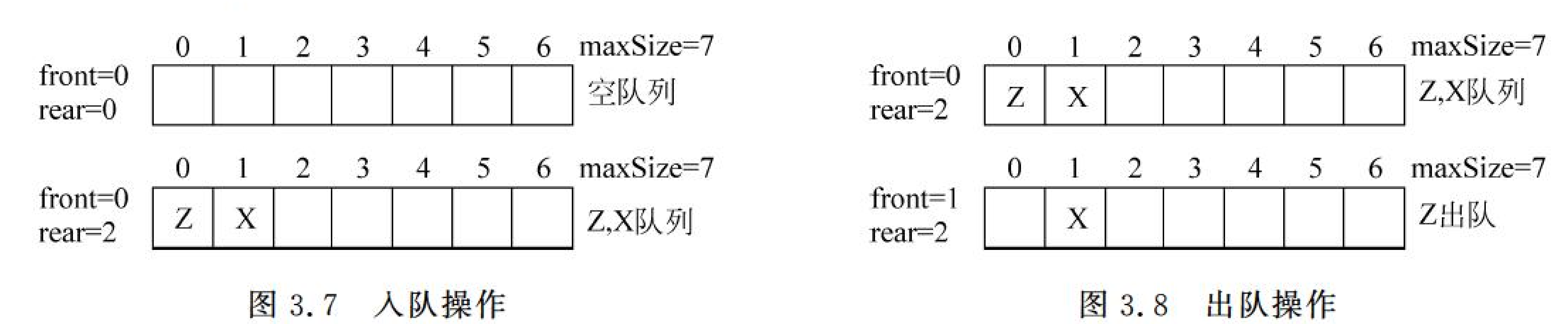


图 3.7 入队操作

图 3.8 出队操作

顺序队列类的 Python 语言描述如下:

```
class SqQueue(IQueue):
    def __init__(self,maxSize):
        self.maxSize = maxSize # 队列的最大存储单元个数
        self.queueElem = [None] * self.maxSize # 队列的存储空间
        self.front = 0 # 指向队首元素
        self.rear = 0 # 指向队尾元素的下一个存储单元

    def clear(self):
        '''将队列置空'''
        self.front = 0
        self.rear = 0

    def isEmpty(self):
        '''判断队列是否为空'''
        return self.rear == self.front

    def length(self):
        '''返回队列的数据元素个数'''
        return self.rear - self.front

    def peek(self):
        '''返回队首元素'''
        if self.isEmpty():
            return None
        else:
            return self.queueElem[self.front]

    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        if self.rear == self.maxSize:
            raise Exception("队列已满")
        self.queueElem[self.rear] = x
        self.rear += 1

    def poll(self):
        '''将队首元素删除并返回其值'''
```

```
        if self.isEmpty():
            return None
        p = self.queueElem[self.front]
        self.front += 1
        return p

    def display(self):
        '''输出队列中的所有元素'''
        for i in range(self.front,self.rear):
            print(self.queueElem[i],end = ' ')
```

2. 循环顺序队列类的描述及实现

分析发现,顺序队列的多次入队和出队操作会造成有存储空间却不能进行入队操作的“假溢出”现象,如图 3.9 所示。顺序队列之所以会出现“假溢出”现象是因为顺序队列的存储单元没有重复使用机制,为了解决顺序队列因数组下标越界而引起的“溢出”问题,可将顺序序列的首尾相连,形成循环顺序队列。循环顺序队列进行入队和出队操作后的状态变化如图 3.10 所示。

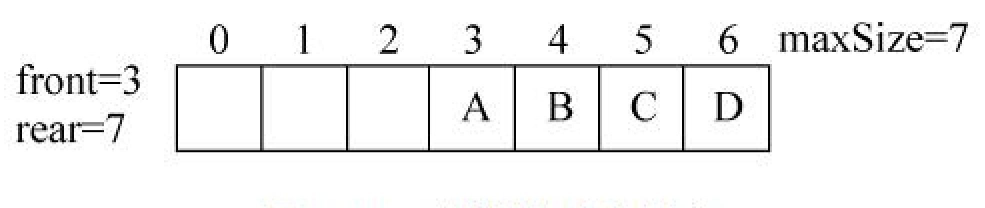


图 3.9 “假溢出”现象

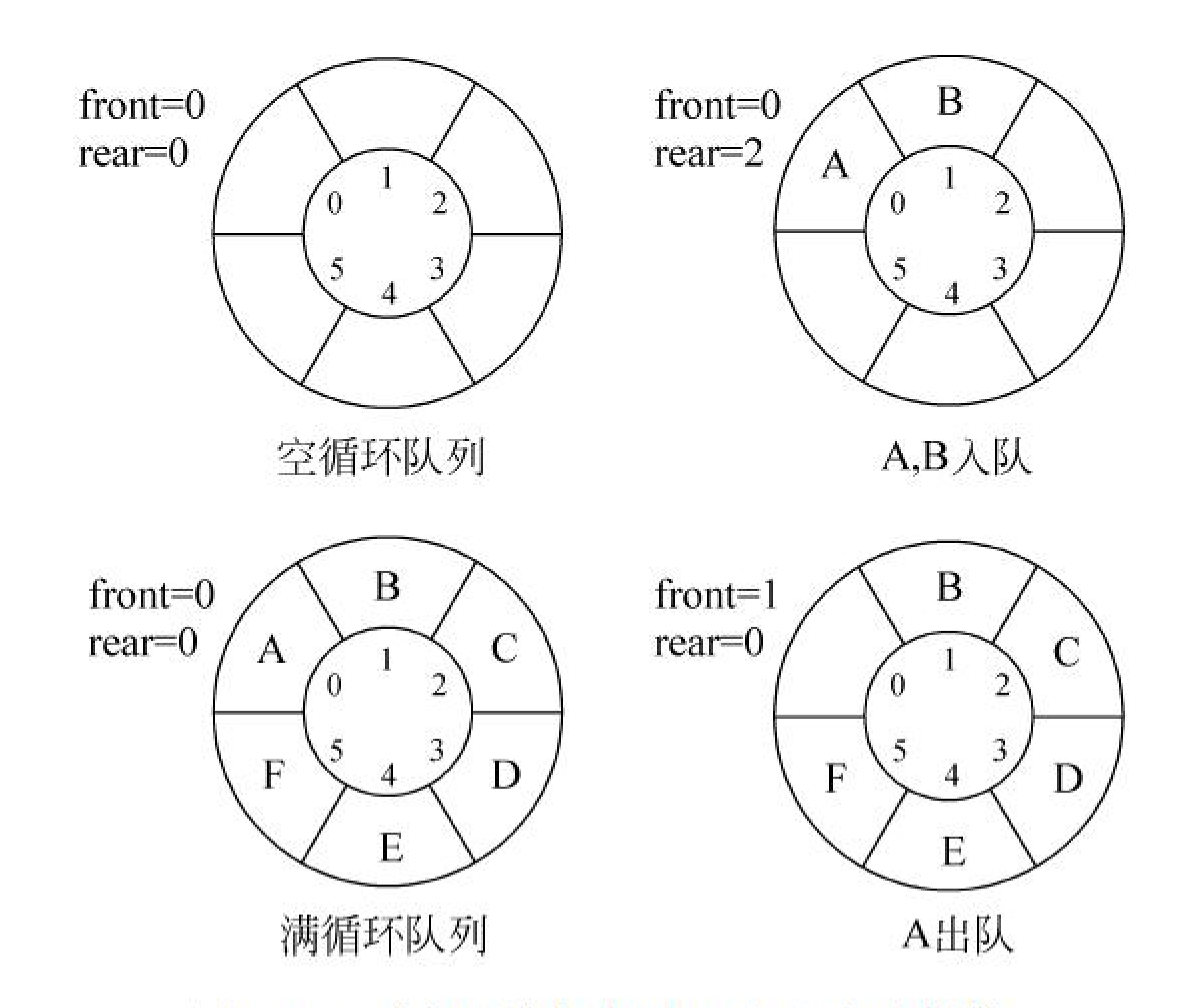


图 3.10 循环顺序队列入队和出队操作

有新的问题产生——队空和队满的判定条件都变为 front==rear,为了解决这一问题,可少利用一个存储单元,队列最多存放 maxSize-1 个数据元素,队空的判定条件为 front==rear,队满的判定条件为 front=(rear+1)%maxSize。

循环顺序队列类和顺序队列类的 Python 语言描述相似,仅是指示变量 front 和 rear 的修改以及队满的判定条件发生了变化。

循环顺序队列的 Python 语言描述如下:

```
class CircleQueue(IQueue):
    def __init__(self,maxSize):
        self.maxSize = maxSize # 队列的最大存储单元个数
```

```
        self.queueElem = [None] * self.maxSize # 队列的存储空间
        self.front = 0 # 指向队首元素
        self.rear = 0 # 指向队尾元素的下一个存储单元

    def clear(self):
        '''将队列置空'''
        self.front = 0
        self.rear = 0

    def isEmpty(self):
        '''判断队列是否为空'''
        return self.rear == self.front

    def length(self):
        '''返回队列的数据元素个数'''
        return (self.rear - self.front + self.maxSize) % self.maxSize

    def peek(self):
        '''返回队首元素'''
        if self.isEmpty():
            return None
        else:
            return self.queueElem[self.front]

    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        if (self.rear + 1) % self.maxSize == self.front:
            raise Exception("队列已满")
        self.queueElem[self.rear] = x
        self.rear = (self.rear + 1) % self.maxSize

    def poll(self):
        '''将队首元素删除并返回其值'''
        if self.isEmpty():
            return None
        p = self.queueElem[self.front]
        self.front = (self.front + 1) % self.maxSize
        return p

    def display(self):
        '''输出队列中的所有元素'''
        i = self.front
        while i!= self.rear:
            print(self.queueElem[i],end = ' ')
            i = (i + 1) % self.maxSize
```

【例 3.5】 假定用于顺序存储一个队列的数组的长度为 N，队首和队尾指针分别为 front 和 rear，写出求此队列长度(即所含元素个数)的公式。

解： 当 rear 大于等于 front 时队列长度为 rear－front，也可以表示为(rear－front＋N)%N；当 rear 小于 front 时队列被分成两个部分，前部分在数组尾部，其元素个数为

$N-1-$front,后部分在数组首部,其元素个数为 rear+1,两者相加为 rear−front+N。综上所述,在任何情况下队列长度的计算公式都为(rear−front+N)%N。

【例 3.6】 在执行操作序列 EnQueue(1)、EnQueue(3)、DeQueue、EnQueue(5)、EnQueue(7)、DeQueue、EnQueue(9)之后队首元素和队尾元素分别是什么? EnQueue(k)表示整数 k 入队,DeQueue 表示队首元素出队。

解:上述操作的执行过程如图 3.11 所示。

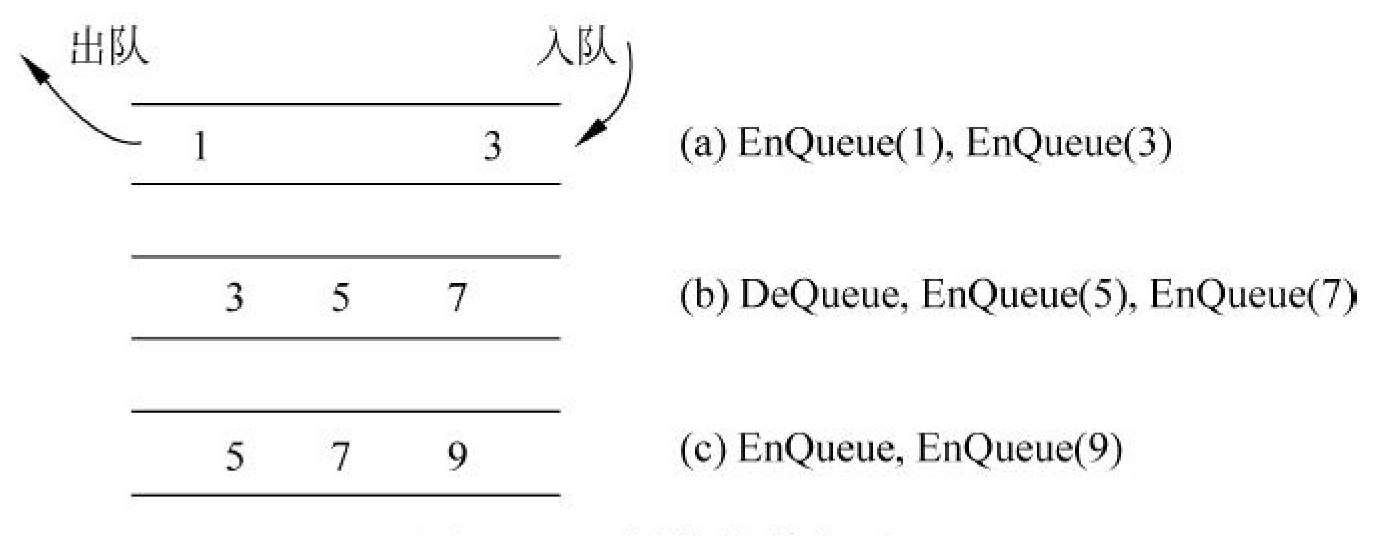


图 3.11 操作的执行过程

所以队首元素为 5,队尾元素为 9。

3.2.4 链队列

链队列用单链表实现,由于入队和出队分别在队列的队尾和队首进行,不存在在队列的任意位置进行插入和删除的情况,所以不需要设置头节点,只需要将指针 front 和 rear 分别指向队首节点和队尾节点,每个节点的指针域指向其后继节点即可。

图 3.12 所示为链队列进行入队操作后的状态变化。

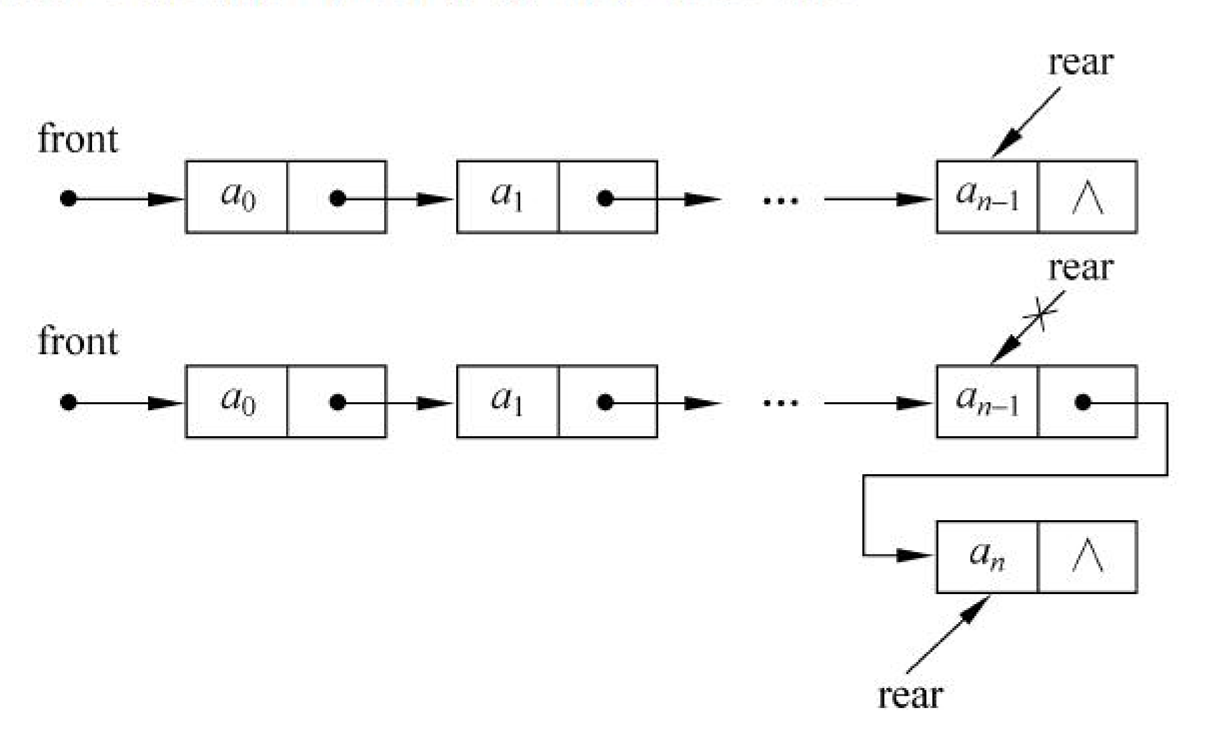


图 3.12 入队操作

图 3.13 所示为链队列进行出队操作后的状态变化。

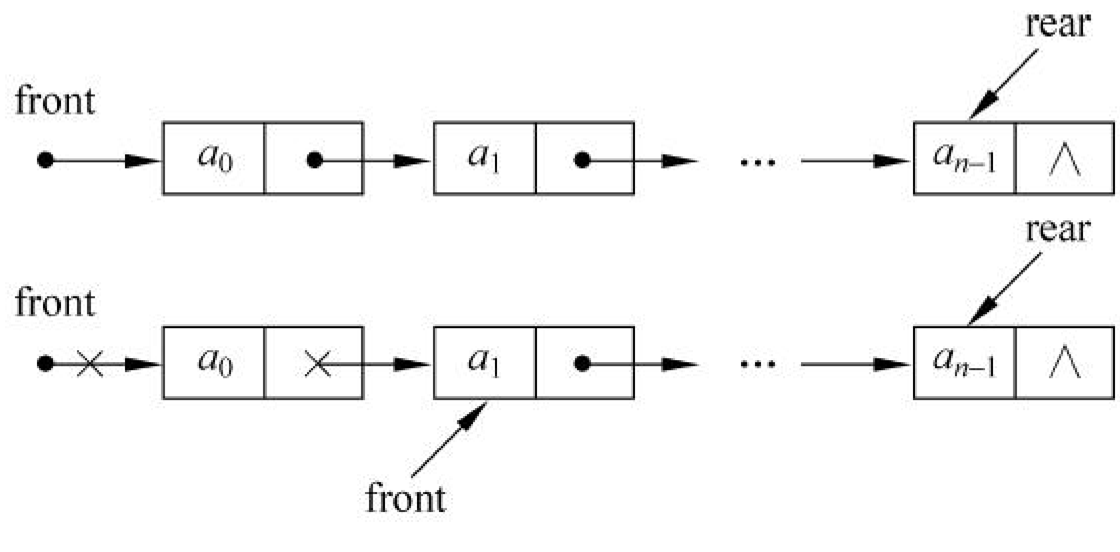


图 3.13 出队操作

利用 Node 类，链队列的 Python 语言描述如下：

```
class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next

class LinkQueue(IQueue):
    def __init__(self):
        self.front = None # 队首指针
        self.rear = None # 队尾指针

    def clear(self):
        '''将队列置空'''
        self.front = None
        self.rear = None

    def isEmpty(self):
        '''判断队列是否为空'''
        return self.front is None

    def length(self):
        '''返回队列的数据元素个数'''
        p = self.front
        i = 0
        while p is not None:
            p = p.next
            i += 1
        return i

    def peek(self):
        '''返回队首元素'''
        if self.isEmpty():
            return None
        else:
            return self.front.data

    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        s = Node(x,None)
        if not self.isEmpty():
            self.rear.next = s
        else:
            self.front = s
        self.rear = s

    def poll(self):
        '''将队首元素删除并返回其值'''
        if self.isEmpty():
            return None
```

```
            p = self.front
            self.front = self.front.next
            if p == self.rear: # 删除节点为队尾节点时需要修改 rear
                self.rear = None
            return p.data

    def display(self):
        '''输出队列中的所有元素'''
        p = self.front
        while p is not None:
            print(p.data,end = ' ')
            p = p.next
```

3.2.5 优先级队列

有些应用中的排队等待问题若仅按照“先来先服务”原则不能满足要求,还需要将任务的重要程度作为排队的依据。例如操作系统中的进程调度管理,每个进程都有一个优先级值表示进程的紧急程度,优先级高的进程先执行,同级进程按照先进先出原则排队等待,因此操作系统需要使用优先级队列来管理和调度进程。又比如打印机的输出任务队列,对于先后到达的打印几百页和几页的任务将需要打印的页数较少的任务先完成,这样使得任务的总的等待时间最小。

优先级队列是在普通队列的基础之上将队列中的数据元素按照关键字的值进行有序排列。优先级队列在队首进行删除操作,但为了保证队列的优先级顺序,插入操作不一定在队尾进行,而是按照优先级插入到队列的合适位置。

和其他数据结构类似,优先级队列也可以采用顺序和链式两种存储结构。但为了快速地访问优先级高的元素以及快速地插入数据元素,通常使用链式存储结构。

1. 优先级队列节点类的描述

```
class PriorityNode:
    def __init__(self,data = None,priority = None,next = None):
        self.data = data # 节点的数据域
        self.priority = priority # 节点的优先级
        self.next = next
```

2. 优先级队列类的描述及实现

```
class PriorityQueue(IQueue):
    def __init__(self):
        self.front = None # 队首指针
        self.rear = None # 队尾指针

    def clear(self):
        '''将队列置空'''
        self.front = None
        self.rear = None

    def isEmpty(self):
```

```
            '''判断队列是否为空'''
            return self.front is None

        def length(self):
            '''返回队列的数据元素个数'''
            p = self.front
            i = 0
            while p is not None:
                p = p.next
                i += 1
            return i

        def peek(self):
            '''返回队首元素'''
            if self.isEmpty():
                return None
            else:
                return self.front.data

        def offer(self,x,priority):
            '''将数据元素 x 插入到队列成为队尾元素'''
            s = PriorityNode(x,priority,None)
            if not self.isEmpty():
                p = self.front
                q = self.front
                while p is not None and p.priority <= s.priority:
                    q = p
                    p = p.next
                # 元素位置的三种情况
                if p is None: # 队尾
                    self.rear.next = s
                    self.rear = s
                elif p == self.front: # 队首
                    s.next = self.front
                    self.front = s
                else: # 队中
                    q.next = s
                    s.next = p
            else:
                self.front = self.rear = s

        def poll(self):
            '''将队首元素删除并返回其值'''
            if self.isEmpty():
                return None
            p = self.front
            self.front = self.front.next
            if p == self.rear: # 删除节点为队尾节点时需要修改 rear
                self.rear = None
            return p.data
```

```
    def display(self):
        '''输出队列中的所有元素'''
        p = self.front
        while p is not None:
            print(p.data,end = ' ')
            p = p.next
```

注意:在此优先队级列中,数据元素的优先级别依据优先数的大小进行判定,即优先数越小优先级别越大。

3. 优先级队列类的应用

【例 3.7】 利用优先级队列模仿操作系统的进程管理问题,要求优先级高的进程先获得 CPU,优先级相同的进程先到的先获得 CPU。假设操作系统中的进程由进程号和进程优先级两部分组成。

解:

```
pq = PriorityQueue()
pq.offer(1,20)
pq.offer(2,30)
pq.offer(3,10)
pq.offer(4,50)
print("进程服务的顺序为: ")
while not pq.isEmpty():
    print(pq.poll())
```

3.3 栈和队列的比较

栈和队列的比较如表 3.1 所示。

表 3.1 栈和队列的比较

相同点	(1) 均为线性结构,数据元素间具有"一对一"的逻辑关系; (2) 都有顺序存储和链式存储两种实现方式; (3) 操作受限,插入操作均在表尾进行(优先级队列除外); (4) 插入与删除操作都具有常数时间
不同点	(1) 栈删除操作在表尾进行,具有后进先出特性;队列的删除操作在表头进行,具有先进先出特性。 (2) 顺序栈可以实现多栈空间共享,而顺序队列则不同

小　　结

(1) 栈是一种特殊的线性表,它只允许在栈顶进行插入和删除操作,具有后进先出的特性,各种运算的时间复杂度为 $O(1)$。栈采用顺序存储结构或者链式存储结构。

(2) 队列是一种特殊的线性表,它只允许在表头进行删除操作、在表尾进行插入操作,具有先进先出的特性,各种运算的时间复杂度为 $O(1)$。队列采用顺序存储结构或者链式存

储结构。

(3) 循环队列是将顺序队列的首尾相连,解决“假溢出”现象的发生。

(4) 优先级队列是在普通队列的基础之上将队列中的数据元素按照关键字的值进行有序排列。在表头进行删除操作,插入操作按照优先级插入到队列的合适位置。

习 题 3

一、选择题

1. 对于栈操作数据的原则是(　　)。

A. 先进先出　　B. 后进先出　　C. 后进后出　　D. 不分顺序

2. 在做入栈运算时应先判别栈是否(①),在做出栈运算时应先判别栈是否(②)。当栈中元素为 n 个,做进栈运算时发生上溢,则说明该栈的最大容量为(③)。

为了增加内存空间的利用率和减少溢出的可能性,由两个栈共享一片连续的内存空间时应将两栈的(④)分别设在这片内存空间的两端,这样当(⑤)时才产生上溢。

①②:A. 空　　B. 满　　C. 上溢　　D. 下溢

③　A. $n-1$　　B. n　　C. $n+1$　　D. $n/2$

④　A. 长度　　B. 深度　　C. 栈顶　　D. 栈底

⑤:　A. 两个栈的栈顶同时到达栈空间的中心点

B. 其中一个栈的栈顶到达栈空间的中心点

C. 两个栈的栈顶在栈空间的某一位置相遇

D. 两个栈均不空,且一个栈的栈顶到达另一个栈的栈底

3. 一个栈的输入序列为 1,2,3,…,n,若输出序列的第一个元素是 n,输出的第 $i(1\leqslant i\leqslant n)$ 个元素是(　　)。

A. 不确定　　B. $n-i+1$　　C. i　　D. $n-i$

4. 若一个栈的输入序列为 1,2,3,…,n,输出序列的第一个元素是 i,则第 j 个输出元素是(　　)。

A. $i-j-1$　　B. $i-j$　　C. $j-i+1$　　D. 不确定的

5. 若已知一个栈的入栈序列是 1,2,3,…,n,其输出序列为 $p_1,p_2,p_3,\cdots,p_n$,若 p_n 是 n,则 p_i 是(　　)。

A. 1　　B. $n-i$　　C. $n-i+1$　　D. 不确定

6. 有 6 个元素 6,5,4,3,2,1 顺序入栈,下列不是合法的出栈序列是(　　)。

A. 5,4,3,6,1,2　　B. 4,5,3,1,2,6

C. 3,4,6,5,2,1　　D. 2,3,4,1,5,6

7. 设栈的输入序列是 1,2,3,4,则(　　)不可能是其出栈序列。

A. 1,2,4,3　　B. 2,1,3,4

C. 1,4,3,2　　D. 4,3,1,2

E. 3,2,1,4

8. 一个栈的输入序列为 1,2,3,4,5,则下列序列中不可能是栈的输出序列的是(　　)。

A. 2,3,4,1,5　　B. 5,4,1,3,2

C. 2,3,1,4,5　　D. 1,5,4,3,2

9. 设一个栈的输入序列是 1,2,3,4,5,则下列序列中是栈的合法输出序列的是(　　)。

A. 5,1,2,3,4　　B. 4,5,1,3,2

C. 4,3,1,2,5　　D. 3,2,1,5,4

10. 某堆栈的输入序列为 a,b,c,d,下面序列中,不可能是它的输出序列的是(　　)。

A. a,c,b,d　　B. b,c,d,a

C. c,d,b,a　　D. d,c,a,b

11. 设 a,b,c,d,e,f 以所给的次序入栈,若在入栈操作时,允许出栈操作,则下面得不到的序列为(　　)。

A. f,e,d,c,b,a　　B. b,c,a,f,e,d

C. d,c,e,f,b,a　　D. c,a,b,d,e,f

12. 设有 3 个元素 X、Y、Z 顺序入栈(入的过程中允许出栈),下列得不到的出栈排列是(　　)。

A. X,Y,Z　　B. Y,Z,X　　C. Z,X,Y　　D. Z,Y,X

13. 输入序列为 A,B,C,变为 C,B,A 时经过的栈操作为(　　)。

A. push,pop,push,pop,push,pop　　B. push,push,push,pop,pop,pop

C. push,push,pop,pop,push,pop　　D. push,pop,push,push,pop,pop

14. 若一个栈以向量 $V[1..n]$存储,初始栈顶指针 top 为 $n+1$,则下面 x 入栈的正确操作是(　　)。

A. top:=top+1;V[top]:=x　　B. V[top]:=x; top:=top+1

C. top:=top−1;V[top]:=x　　D. V[top]:=x; top:=top−1

15. 若栈采用顺序存储方式存储,现两栈共享空间 $V[1..m]$,top[i]代表第 i 个栈(i=1,2)的栈顶,栈 1 的底在 $V[1]$、栈 2 的底在 $V[m]$,则栈满的条件是(　　)。

A. |top[2]−top[1]|=0　　B. top[1]+1=top[2]

C. top[1]+top[2]=m　　D. top[1]=top[2]

二、填空题

1. 向量、栈和队列都是________结构,可以在向量的________位置插入和删除元素;对于栈只能在________插入和删除元素;对于队列只能在________插入和________删除元素。

2. 栈是一种特殊的线性表,允许插入和删除运算的一端称为________,不允许插入和删除运算的一端称为________。

3. ________是被限定为只能在表的一端进行插入运算、在表的另一端进行删除运算的线性表。

4. 在一个循环队列中,队首指针指向队首元素的________位置。

5. 在具有 n 个单元的循环队列中,队满时共有________个元素。

6. 向栈中压入元素的操作是先________,后________。

7. 从循环队列中删除一个元素，其操作是先________，后________。

8. 顺序栈用 data[1..n]存储数据，栈顶指针是 top，则值为 x 的元素入栈的操作是________。

9. 表达式 23＋[(12×3－2)/4＋34×5/7]＋108/9 的后缀表达式是________。

10. 引入循环队列的目的是为了克服________。

三、算法设计题

1. 把十进制整数转换为二至九进制数并输出。

2. 堆栈在计算机语言的编译过程中用来进行语法检查，试编写一个算法检查一个字符串中的花括号、方括号和圆括号是否配对，若能够全部配对则返回逻辑真，否则返回逻辑假。

3. 斐波那契(Fibonacci)数列的定义为它的第 1 项和第 2 项分别为 0 和 1，以后各项为其前两项之和。若斐波那契数列中的第 n 项用 Fib(n)表示，则计算公式如下：

$$\mathrm{Fib}(n)=n-1\ (n=1 \text{ 或 } 2)$$

$$\mathrm{Fib}(n)=\mathrm{Fib}(n-1)+\mathrm{Fib}(n-2)\ (n>2)$$

试编写出计算 Fib(n)的递归算法和非递归算法。

第4章 串和数组

4.1 串

4.1.1 串的基本概念

字符串也叫串,是由字符组成的有限序列,是一种常用的非数值数据。串的逻辑结构是线性表,串是一种特殊的线性表,其每个数据元素都是一个字符。串的操作特点与线性表不同,主要是对子串进行操作,通常采用顺序存储结构存储。

字符串可以表示为 str="$a_0a_1\cdots a_i\cdots a_{n-1}$",其中 str 为串名,也叫串变量;$i$ 为字符 a_i 在串中的位序号;双引号中的字符序列称为串值;n 为串的长度。当 $n=0$ 时字符串不包含任何字符,为空串;当字符串由一个或多个空白字符组成时为空白串。

字符串中任意个连续字符组成的子序列称为字符串的子串,此字符串为该子串的主串。子串在主串中的位置是指子串在主串中第一次出现时的第一个字符在主串中的位置。空串是任意串的子串。每个字符串都是其自身的子串,除自身外,主串的其他子串称为主串的真子串。

串的比较规则和字符的比较规则有关,字符的比较规则由所属的字符集的编码决定。两个串相等是指串长度相同并且各对应位置上的字符也相同。两个串的大小由对应位置上的首个不同字符的大小决定,字符比较次序是从头开始依次向后。当两个串的长度不等而对应位置上的字符都相同时较长的串定义为较大。

4.1.2 串的抽象数据类型描述

字符串是数据元素类型为字符的线性表,其抽象数据类型描述与线性表相似,又根据串在实际问题中的应用抽象出串的基本操作,可得串的抽象数据类型 Python 语言描述如下:

```python
from abc import ABCMeta,abstractmethod,abstractproperty

class IString(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将字符串置成空串'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断是否为空串'''
```

```python
        pass
    @abstractmethod
    def length(self):
        '''返回串的长度'''
        pass
    @abstractmethod
    def charAt(self,i):
        '''读取并返回串中的第 i 个数据元素'''
        pass
    @abstractmethod
    def subString(self,begin,end):
        '''返回位序号从 begin 到 end-1 的子串'''
        pass
    @abstractmethod
    def insert(self,i,str):
        '''在第 i 个字符之前插入子串 str'''
        pass
    @abstractmethod
    def delete(self,begin,end):
        '''删除位序号从 begin 到 end-1 的子串'''
        pass
    @abstractmethod
    def concat(self,str):
        '''将 str 连接到字符串的后面'''
        pass
    @abstractmethod
    def compareTo(self,str):
        '''比较 str 和当前字符串的大小'''
        pass
    @abstractmethod
    def indexOf(self,str,begin):
        '''从位序号为 begin 的字符开始搜索与 str 相等的子串'''
        pass
```

字符串的抽象数据类型 Python 抽象类包含了串的主要基本操作，如果要使用这个接口，还需要具体的类来实现。串的 Python 抽象类的实现方法主要有以下两种。

(1) 基于顺序存储的实现，为顺序串。

(2) 基于链式存储的实现，为链串。

4.1.3 顺序串

1. 顺序串类的描述

顺序串与顺序表的逻辑结构相同，存储结构类似，均可用数组来存储数据元素。但串具有独特的不同于线性表的操作，属于特殊类型的线性表。图 4.1 所示为顺序串。

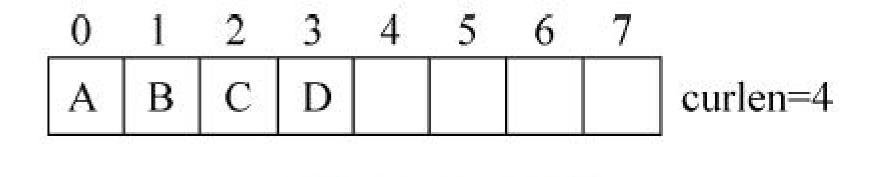


图 4.1 顺序串

实现 IString 抽象类的顺序串类的 Python 语言描述如下：

```
class SqString(IString):
    def __init__(self,obj = None):
        if obj is None: # 构造空串
            self.strValue = [] # 字符数组存放串值
            self.curLen = 0 # 当前串的长度
        elif isinstance(obj,str): # 以字符串构造串
            self.curLen = len(obj)
            self.strValue = [None] * self.curLen
            for i in range(self.curLen):
                self.strValue[i] = obj[i]
        elif isinstance(obj,list): # 以字符列表构造串
            self.curLen = len(obj)
            self.strValue = [None] * self.curLen
            for i in range(self.curLen):
                self.strValue[i] = obj[i]

    def clear(self):
        '''将字符串置成空串'''
        self.curLen = 0

    def isEmpty(self):
        '''判断是否为空串'''
        return self.curLen == 0

    def length(self):
        '''返回串的长度'''
        return self.curLen

    def charAt(self,i):
        '''读取并返回串中的第 i 个数据元素'''
        if i<0 or i>=self.curLen:
            raise IndexError("String index out of range")
        return self.strValue[i]

    def allocate(self,newCapacity):
        '''将串的长度扩充为 newCapacity'''
        tmp = self.strValue
        self.strValue = [None] * newCapacity
        for i in range(self.curLen):
            self.strValue[i] = tmp[i]

    def subString(self,begin,end):
        '''返回位序号从 begin 到 end-1 的子串'''
        pass

    def insert(self,i,str):
        '''在第 i 个字符之前插入子串 str'''
        pass
```

```python

    def delete(self,begin,end):
        '''删除位序号从 begin 到 end - 1 的子串'''
        pass

    def concat(self,str):
        '''将 str 连接到字符串的后面'''
        pass

    def compareTo(self,str):
        '''比较 str 和当前字符串的大小'''
        pass

    def indexOf(self,str,begin):
        '''从位序号为 begin 的字符开始搜索与 str 相等的子串'''
        pass

    def display(self):
        '''打印字符串'''
        for i in range(self.curLen):
            print(self.strValue[i],end = '')
```

2. 顺序串基本操作的实现

1) 求子串操作

求子串操作 subString(begin,end)是返回长度为 n 的字符串中位序号从 begin 到 end-1 的字符序列,其中 $0\leqslant \text{begin}\leqslant n-1$,$\text{begin}<\text{end}\leqslant n$。其主要步骤如下。

(1) 检查参数 begin 和 end 是否满足 $0\leqslant \text{begin}\leqslant n-1$ 和 $\text{begin}<\text{end}\leqslant n$,若不满足,抛出异常。

(2) 返回位序号为 begin 到 end-1 的字符序列。

【算法 4.1】 求子串操作。

```python
def subString(self,begin,end):
    '''返回位序号从 begin 到 end - 1 的子串'''
    if begin < 0 or begin > = end or end > self.curLen:
        raise IndexError("参数不合法")
    tmp = [None] * (end - begin)
    for i in range(begin,end):
        tmp[i - begin] = self.strValue[i] # 复制子串
    return SqString(tmp)
```

2) 插入操作

插入操作 insert(i,str)是在长度为 n 的字符串的第 i 个元素之前插入串 str,其中 $0\leqslant i\leqslant n$。其主要步骤如下:

(1) 判断参数 i 是否满足 $0\leqslant i\leqslant n$,若不满足,则抛出异常。

(2) 重新分配存储空间为 $n+m$,m 为插入的字符串 str 的长度。

(3) 将第 i 个及之后的数据元素向后移动 m 个存储单元。

(4) 将 str 插入到字符串从 i 开始的位置。

【算法 4.2】 插入操作。

```
def insert(self,i,str):
    '''在第 i 个字符之前插入子串 str'''
    if i < 0 or i > self.curLen:
        raise IndexError("插入位置不合法")
    length = str.length()
    newCapacity = self.curLen + length
    self.allocate(newCapacity)
    for j in range(self.curLen - 1,i - 1, - 1):
        self.strValue[j + length] = self.strValue[j]
    for j in range(i,i + length):
        # print(j - i,str.charAt(j - i))
        self.strValue[j] = str.charAt(j - i)
    self.curLen = newCapacity
```

3) 删除操作

删除操作 delete(begin,end)是将长度为 n 的字符串的位序号为 begin 到 end－1 的元素删除,其中参数 begin 和 end 满足 0≤begin≤curLen－1 和 begin＜end≤curLen。其主要步骤如下。

(1) 判断参数 begin 和 end 是否满足 0≤begin≤curLen－1 和 begin＜end≤curLen,若不满足,则抛出异常。

(2) 将字符串位序号为 end 的数据元素及其之后的数据元素向前移动到位序号为 begin 的位置。

(3) 字符串长度减小 end－begin。

【算法 4.3】 删除操作。

```
def delete(self,begin,end):
    '''删除位序号从 begin 到 end - 1 的子串'''
    if begin < 0 or begin > = end or end > self.curLen:
        raise IndexError("参数不合法")
    for i in range(begin,end):
        self.strValue[i] = self.strValue[i + end - begin]
    self.curLen = self.curLen - end + begin
```

4) 连接操作

concat(str)是将串 str 插入到字符串的尾部,此时调用 insert(curLen,str)即可实现。

5) 比较操作

比较操作 compareTo(str)是将字符串与串 str 按照字典序进行比较。若当前字符串较大,返回 1;若相等,返回 0。若当前字符串较小,返回－1。其主要步骤如下。

(1) 确定需要比较的字符的个数 n 为两个字符串长度的较小值。

(2) 从下标 0 至 $n-1$ 依次进行比较。

【算法 4.4】 比较操作。

```
def compareTo(self,str):
    '''比较 str 和当前字符串的大小'''
```

```
        n = self.curLen if self.curLen < str.length() else str.length()
        for i in range(n):
            if self.strValue[i]> str.charAt(i):
                return 1
            if self.strValue[i]< str.charAt(i):
                return -1
        if self.curLen > str.length():
            return 1
        elif self.curLen < str.length():
            return -1
        return 0
```

【例 4.1】 编写一个程序,完成构造串、判断串是否为空、返回串的长度、求子串等操作。

```
s1 = SqString(['1', '2', '3', '4', '5', '6'])
s2 = SqString(['1', '2', '3', '4', '5', '6'])
s1.insert(1,s2)
s1.display()
print()
s1.delete(1,6)
s1.display()
print()
s1.concat(s2)
s1.display()
print()
s3 = s1.subString(1,6)
s3.display()
print()
print(s1.compareTo(s3))
```

4.1.4 链串

链串采用链式存储结构,和线性表的链式存储结构类似,可以采用单链表存储串值。链串由一系列大小相同的节点组成,每个节点用数据域存放字符,指针域存放指向下一个节点的指针。

与线性表不同的是每个节点的数据域可以是一个字符或者多个字符。若每个节点的数据域为一个字符,这种链表称为单字符链表;若每个节点的数据域为多个字符,则称为块链表。在块链表中每个节点的数据域不一定被字符占满,可通过添加空字符或者其他非串值字符来简化操作。图 4.2 所示为两种不同类型的链串。

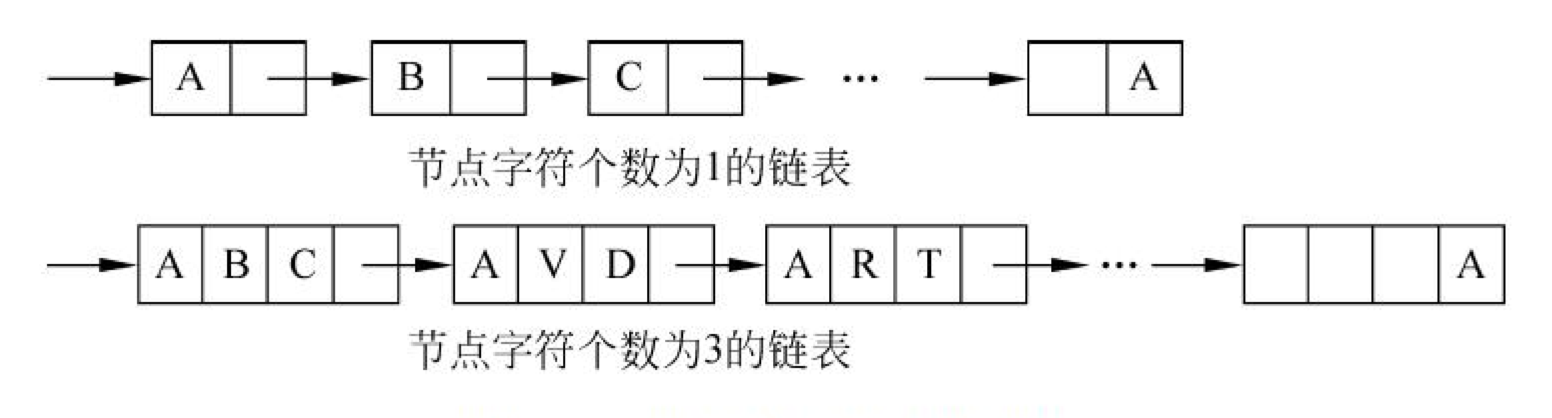


图 4.2 链串的两种存储结构

在串的链式存储结构中,单字符链表的插入、删除操作较为简单,但存储效率低。块链表虽然存储效率较高但插入、删除操作需要移动字符,较为复杂。此外,与顺序串相比,链串需要从头部开始遍历才能访问某个位置的元素。

用户在应用中需要根据实际情况选择合适的存储结构。

4.2 串的模式匹配

串的模式匹配也叫查找定位,指的是在当前串中寻找模式串的过程,主要的模式匹配算法有 Brute Force 算法和 KMP 算法。

4.2.1 Brute Force 算法

Brute Force 算法:从主串的第一个字符开始和模式串的第一个字符进行比较,若相等,则继续比较后续字符;否则从主串的第二个字符开始重新和模式串进行比较。以此类推,直到模式串的每个字符依次与主串的字符相等,匹配成功。

【算法 4.5】 Brute Force 模式匹配。

```
def BF(self,str,begin):
    if str.length()<= self.curLen and str is not None and self.curLen > 0:
        i = begin
        length = str.length()
        while(i <= self.curLen - length):
            for j in range(length):
                if str.charAt(j)!= self.strValue[j + i]:
                    i += 1
                    break
                elif j == length - 1:
                    return i
    return -1
```

Brute Force 算法的实现简单,但效率非常低。m 为模式串的长度,n 为主串的长度。

(1) 最好情况:第一次匹配即成功,比较次数为模式串的长度 m,时间复杂度为 $O(m)$。

(2) 最坏情况:每次匹配比较至模式串的最后一个字符,并且比较了目标串中所有长度为 m 的子串,此时的时间复杂度为 $O(m\times n)$。

这是因为 Brute Force 算法是一种带回溯的模式匹配算法,它将目标串中所有长度为 m 的子串依次与模式串进行匹配,若主串和模式串已有多个字符相同,有一个不同的字符出现,就需要将主串的开始比较位置增加 1 后与整个模式串再次重新比较,这样没有丢失任何匹配的可能。但是每次匹配没有利用前一次匹配的比较结果,使算法中存在较多的重复比较,降低了算法的效率;如果利用部分字符匹配的结果,可将算法的效率提高。因此提出了 KMP 算法,在下一节进行介绍。

4.2.2 KMP 算法

KMP 算法的主要思想是当某次匹配失败时主串的开始比较位置不回退,而是利用部分字符匹配的结果将模式串向后移动较远的距离后再继续进行比较。

1. KMP 模式匹配算法分析

设主串为 s="ababcabdabcabca"、模式串为 p="abcabc"，指针 i、j 分别指示主串和模式串所比较字符的位序号。

(1) 在第一趟匹配中，当 $s_0=p_0$、$s_1=p_1$、$s_2\neq p_2$时 $i=2$、$j=2$。

(2) 在第二趟匹配中应修改 $i=1$、$j=0$ 后再次进行比较。但由于 $p_0\neq p_1$、$s_1=p_1$，所以 $s_1\neq p_0$，故此时不需要进行 s_1 和 p_0 的比较，而只需比较 s_2 和 p_0。

(3) 在第三趟匹配中，当 $s_7\neq p_5$ 时 $i=7$、$j=5$，此时有 $s_2s_3s_4s_5s_6=p_0p_1p_2p_3p_4$，因为 $p_0\neq p_1$、$p_0\neq p_2$，所以以 s_3 和 s_4 为开始位置的比较不必进行。又因为 $p_0p_1=p_3p_4$，所以 $s_5s_6=p_0p_1$，这两次比较也可以省略。

通过对模式串匹配过程的分析可以发现，从模式串本身即可计算出匹配失败后下一次匹配模式串的比较位置，主串的比较位置不需要进行回退。

设主串为 s="aba bcabdabcabca"、模式串为 p="abcabc"，指针 i、j 分别指示主串和模式串所比较字符的位序号。当某次匹配不成功时有 $s_i\neq p_j$，并且 $s_{i-j}s_{i-j+1}\cdots s_{i-1}=p_0p_1\cdots p_{j-1}$。此时需要寻找前缀子串 $p_0p_1\cdots p_{k-1}=p_{j-k}p_{j-k+1}\cdots p_{j-1}$，其中 $0<k<j$，这时候即满足 $s_{i-k}s_{i-k+1}\cdots s_{i-1}=p_0p_1\cdots p_{k-1}$，下一次匹配可直接比较 s_i 和 p_k。此外，为了减少比较次数，k 应取最大值，即 $p_0p_1\cdots p_{k-1}$ 应为满足此性质的最长前缀子串。若 k 不存在，下一次匹配则直接比较 s_i 和 p_0。

2. k 值的计算

通过前面的分析已知，每次模式串开始比较的位置（即 k 的值）仅与模式串本身有关。一般用 next[j]来表示 p_j 对应的 k 值。

初始时可定义 next[0]=−1，next[1]=0。

设 next[j]=k，则 $p_0p_1\cdots p_{k-1}=p_{j-k}p_{j-k+1}\cdots p_{j-1}$，$k$ 为满足等式的最大值。计算 next[$j+1$]的值。

(1) 若 $p_k=p_j$，则存在 $p_0p_1\cdots p_{k-1}p_k=p_{j-k}p_{j-k+1}\cdots p_{j-1}p_j$，此时 next[$j+1$]=$k+1$。

(2) 若 $p_k\neq p_j$，可以把计算 next[j]的值的问题看成新的模式匹配过程，主串为 p，模式串为 p 的前缀子串。

出现不匹配，应将模式串的比较位置变为 k'=next[k]，若 $p_j=p_{k'}$，则 next[$j+1$]=$k'+1$=next[k]+1，否则继续执行步骤(2)，直到 $p_j=p_k$，或者当 $k=0$ 并且 $p_j\neq p_k$时 next[$j+1$]=0。

【算法 4.6】 求解 next[j]。

```
def next(p):
    next = [0] * p.length() # next 数组
    k = 0 # 模式串指针
    j = 1 # 主串指针
    next[0] = -1
    next[1] = 0
    while j < p.length() - 1:
        if p.charAt(j) == p.charAt(k):
            next[j + 1] = k + 1
            k += 1
            j += 1
        elif k == 0:
```

```
                next[j+1] = 0
                j += 1
            else:
                k = next[k]
        return next
```

3. KMP 算法步骤

KMP 算法的主要步骤如下。

(1) 计算模式串的 next[]函数值。

(2) i 为主串的比较字符位序号,j 为模式串的比较字符位序号。当字符相等时,i、j 分别加 1 后继续比较;否则 i 的值不变,j=next[j],继续比较。

(3) 重复步骤(2),直到 j 等于模式串的长度时匹配成功,否则匹配失败。

【算法 4.7】 KMP 算法。

```
def KMP(self,p,begin):
    next = SqString.next(p) # 计算 next 值
    i = begin # # i 为主串的字符指针
    j = 0
    while i < self.curLen and j < p.length():
        if j ==- 1 or self.strValue[i] == p.charAt(j):
            # 比较的字符相等或者比较主串的下一个字符
            i += 1
            j += 1
        else:
            j = next[j]
    if j == p.length():
        return i - j # 匹配
    else:
        return - 1
```

设主串的长度为 n、模式串的长度为 m,求 next[]的时间复杂度为 $O(m)$。在 KMP 中,因主串的下标不需要回退,比较次数最多为 $n-m+1$,所以 KMP 算法的时间复杂度为 $O(m+n)$。

【例 4.2】 求字符串 str="abcababc"的 next[j]的值。

解:

```
p= SqString('abcababc')
print(SqString.next(p))
```

当 $j=0$ 时,next[0]=−1;

当 $j=1$ 时,next[1]=0;

当 $j=2$ 时,next[2]=0;

当 $j=3$ 时,next[3]=0;

当 $j=4$ 时,next[4]=1;

当 $j=5$ 时,next[5]=2;

当 $j=6$ 时,next[6]=1;

当 $j=7$ 时,next[7]=2。

【例 4.3】 设计程序,分别统计模式匹配的 BF 算法和 KMP 算法的比较次数。主串为

s="abcabcccabc",模式串为 t="bcc"。

解:分别对 Brute Force 算法和 KMP 算法进行修改,额外返回比较的次数,即在循环中增加计数变量 count+=1。

```
def BF(self,str,begin):
    count = 0
    if str.length()<= self.curLen and str is not None and self.curLen> 0:
        i = begin
        length = str.length()
        while(i<= self.curLen - length):
            for j in range(length):
                count += 1
                if str.charAt(j)!= self.strValue[j + i]:
                    i += 1
                    break
                elif j == length - 1:
                    return i,count
    return - 1,count

def next(p):
    next = [0] * p.length() # next 数组
    k = 0 # 模式串指针
    j = 1 # 主串指针
    next[0] = -1
    next[1] = 0
    while j< p.length() - 1:
        if p.charAt(j) == p.charAt(k):
            next[j + 1] = k + 1
            k += 1
            j += 1
        elif k == 0:
            next[j + 1] = 0
            j += 1
        else:
            k = next[k]
    return next

def KMP(self,p,begin):
    count = 0
    next = SqString.next(p) # 计算 next 值
    i = begin # # i 为主串的字符指针
    j = 0
    while i < self.curLen and j < p.length():
        count += 1
        if j ==- 1 or self.strValue[i] == p.charAt(j):
            # 比较的字符相等或者比较主串的下一个字符
            i += 1
            j += 1
        else:
            j = next[j]
```

```
        if j == p.length():
            return i - j,count # 匹配
        else:
            return - 1,count
```

再输出比较次数：

```
s = SqString('abcabcccabc')
t = SqString('bcc')
print(s.BF(t,0)[1])
print(s.KMP(t,0)[1])
```

4.3 数　　组

4.3.1 数组的基本概念

数组是 n 个具有相同数据类型的数据元素构成的集合，数组元素按某种次序存储在地址连续的存储单元中，是顺序存储的随机存储结构。

数组元素在数组中的位置称为数组元素的下标，用户通过下标可以访问相应的数组元素。数组下标的个数是数组的维数，具有一个下标的数组叫一维数组，具有两个下标的数组叫二维数组。一维数组的逻辑结构是线性表，多维数组是线性表的扩展。二维数组可以看成数组元素是一维数组的数组。图 4.3 所示为二维数组的矩阵表示。

$$A_{m\times n}=\begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图 4.3　二维数组的矩阵表示

二维数组中的每个数据元素 $a_{i,j}$ 都受到两个关系的约束，即行关系和列关系。$a_{i,j+1}$ 是 $a_{i,j}$ 在行关系中的后继元素；$a_{i+1,j}$ 是 $a_{i,j}$ 在列关系中的后继元素。

因为二维数组可以看成数组元素是一维数组的数组，所以二维数组也可看成线性表，即 $A=(a_0,a_1,\cdots,a_{n-1})$，其中每个数据元素 a_i 是一个列向量的线性表，即 $a_i=(a_{0,i},a_{1,i},\cdots,a_{m-1,i})$；或者表述为 $A=(a_0,a_1,\cdots,a_{m-1})$，其中每个数据元素 a_i 是一个行向量的线性表，即 $a_i=(a_{0,i},a_{1,i},\cdots,a_{n-1,i})$。其中，每个元素同时属于两个线性表，第 i 行的线性表和第 j 列的线性表，具体可以分析如下。

(1) $a_{0,0}$ 是起点，没有前驱元素；$a_{m-1,n-1}$ 是终点，没有后继元素。

(2) 边界元素 $a_{i,0}$ 和 $a_{0,j}$ $(1\leqslant j<n,1\leqslant i<m)$ 只有一个前驱元素；$a_{i,n-1}$ 和 $a_{m-1,j}$ $(0\leqslant j<n-1,1\leqslant i<m-1)$ 只有一个后继元素。

(3) $a_{i,j}$ $(1\leqslant j<n-1,1\leqslant i<m-1)$ 有两个前驱元素和两个后继元素。

4.3.2 数组的特性

数组元素被存放在一组地址连续的存储单元里，并且每个数据元素的大小相同，故只要已知首地址和每个数据元素占用的内存单元大小即可求出数组中任意数据元素的存储地址。

对于一维数组 $A[n]$，数据元素的存储地址为 $\mathrm{Loc}(i)=\mathrm{Loc}(0)+i\times L(0\leqslant i<n)$，其中 $\mathrm{Loc}(i)$ 是第 i 个元素的存储地址，$\mathrm{Loc}(0)$ 是数组的首地址，L 是每个数据元素占用的字

节数。

对于二维数组，采用行优先顺序进行存储，即先存储数组的第一行，再依次存储其他各行。对于一个 $n \times m$ 的数组 A[n][m]，数组元素的存储地址为 $\text{Loc}(i,j)=\text{Loc}(0,0)+(i\times m+j)\times L$，其中 $\text{Loc}(i,j)$ 是第 i 行第 j 列的数组元素的存储地址，Loc(0,0)是数组的首地址，L 是每个数据元素占用的字节数。

将计算数组元素的存储位置的公式推广到一般情况，可得 n 维数组 A[m_1][m_2]…[m_n]的数据元素的存储位置：

$$
\begin{aligned}
&\text{Loc}(i_1,i_2,\cdots,i_n)\\
&=\text{Loc}(0,0,\cdots,0)+(i_1\times m_2\times\cdots\times m_n+i_2\times m_3\times\cdots\times m_n+i_{n-1}\times m_n+i_n)\times L\\
&=\text{Loc}(0,0,\cdots,0)+\left(\sum_{j=1}^{n} i_j\prod_{k=j+1}^{n} m_k+i_n\right)\times L
\end{aligned}
$$

在 n 维数组中，计算数组中数据元素的存储地址的时间复杂度为 $O(1)$，n 维数组是一种随机存储结构。

4.3.3 数组的遍历

对二维数组进行遍历操作有两种次序，即行主序和列主序。

(1) 行主序：以行序为主要次序，按行序递增访问数组的每行，同一行按列序递增访问数组元素。

(2) 列主序：以列序为主要次序，按列序递增访问数组的每列，同一列按行序递增访问数组元素。

【例 4.4】 设计算法，求二维数组 A[n,n]的两条对角线元素之和。

解：

```
def sumOfDiagonal(a):
    n = len(a[0])
    sum1 = sum2 = 0
    for i in range(n):
        sum1 += a[i][i]
        sum2 += a[i][n - i - 1]
    sum = sum1 + sum2
    if n % 2 == 1:
        sum -= a[n//2][n//2]
    return sum
```

4.4 特殊矩阵的压缩存储

在科学技术和工程计算的许多领域，矩阵是数值分析问题研究的对象。特殊矩阵是具有许多相同数据元素或者零元素且数据元素的分布具有一定规律的矩阵，例如对称矩阵、三角矩阵和对角矩阵。

数据压缩技术是计算机软件领域研究的一个重要问题，图像、音频、视频等多媒体信息都需要进行数据压缩存储。本节将以特殊矩阵为例介绍矩阵的压缩存储。

矩阵采用二维数组进行存储,至少占用 $m\times n$ 个存储单元。当矩阵的阶数很大时,矩阵所占用的存储空间巨大,因此需要研究矩阵的压缩存储问题,根据不同矩阵的特点设计不同的压缩存储方法,节省存储空间,同时保证采用压缩存储的矩阵仍然能够正确地进行各种矩阵运算。

常用的矩阵压缩存储方法主要有以下两种。

(1) 对于零元素分布有规律的特殊矩阵,采用线性压缩或三角形的二维数组,只存储有规律的部分元素。

(2) 对于零元素分布没有规律的特殊矩阵,只存储非零元素。

4.4.1 三角矩阵的压缩存储

三角矩阵包括上三角矩阵和下三角矩阵。假如是一个 n 阶矩阵,由 $n(n+1)/2$ 个元素组成。若当 $i<j$ 时矩阵中的数据元素满足 $=0$,矩阵为下三角矩阵;若当 $i>j$ 时,矩阵中的数据元素满足 $=0$,矩阵为上三角矩阵。

三角矩阵中具有近一半的分布有规律的零元素,所以三角矩阵采取只存储主对角线以及上或下三角部分的矩阵元素的压缩方法,主要分为以下两种。

1. 线性压缩存储

将下三角矩阵的主对角线及其以下元素按行主序顺序压缩成线性存储结构,存储元素的个数为 $n(n+1)/2$,其中元素的存储地址如下:

$$k=\begin{cases}\left(\dfrac{i(i+1)}{2}+j\right)*L,(i\geqslant j)\\ \text{空},(i<j)\end{cases}$$

L 为数据元素所占据存储空间的字节数。

计算各数据元素的存储地址的时间复杂度为 $O(1)$,三角矩阵的线性压缩存储结构是随机存储结构。

2. 使用三角形的二维数组压缩存储

三角形的二维数组实际上是一种动态数组结构,第 i 行一维数组的长度为 $i+1$,存储在 mat[i][j]中,如图 4.4 所示。计算各数据元素的存储地址的时间复杂度为 $O(1)$,此压缩存储结构是随机存储结构。

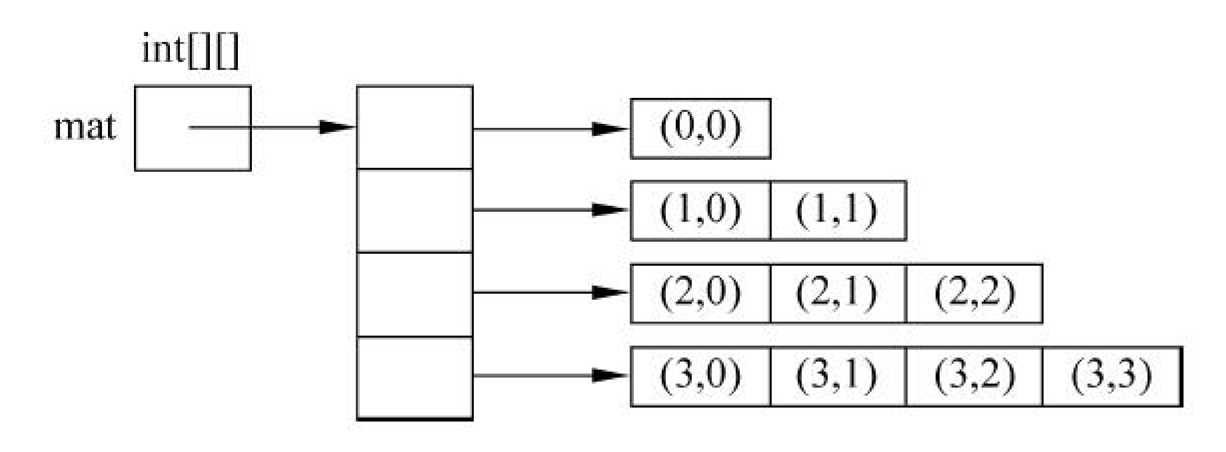


图 4.4 下三角矩阵的三角形二维数组的压缩存储结构

4.4.2 对称矩阵的压缩存储

n 阶对称矩阵是指一个 n 阶矩阵中的数据元素满足 $a_{i,j}=a_{j,i}$。对称矩阵在进行压缩存储时只存储主对角线和上或下部分数据元素,即将对称矩阵的主对角线及其上或下部分数

据元素按行主序顺序压缩成线性存储，占用 $n(n+1)/2$ 个存储单元，矩阵元素的线性压缩存储地址为：

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j(i\geqslant j)\\\dfrac{j(j+1)}{2}+i(i<j)\end{cases}$$

4.4.3 对角矩阵的压缩存储

如果一个矩阵的所有非零元素都集中在以主对角线为中心的带状区域，则称该矩阵为对角矩阵。它是一个 n 阶矩阵，除了主对角线上的元素，其他元素均为 0，则是主对角矩阵；除了主对角线上及主对角线上下各一个元素外，其余元素均为 0，为三对角矩阵。

在压缩存储对角矩阵时，只存储主对角线及其两侧部分的元素。如压缩存储主对角矩阵，将主对角元素顺序压缩成线性存储，存储元素个数为 n，矩阵数据元素的线性压缩存储地址为：

$$k=i \quad 或 \quad k=j$$

4.4.4 稀疏矩阵的压缩存储

稀疏矩阵是指矩阵中的非零元素个数远远小于矩阵元素个数并且非零元素的分布没有规律的矩阵。设矩阵中有 t 个非零元素，非零元素占元素总数的比例称为矩阵的稀疏因子，通常稀疏因子小于 0.05 的矩阵称为稀疏矩阵。一般使用以下几种方法进行稀疏矩阵的压缩存储。

1. 稀疏矩阵的非零元素三元组

稀疏矩阵的压缩存储原则是只存储矩阵中的非零元素，而仅存储非零元素是不够的，必须存储该元素在矩阵中的位置。矩阵元素的行号、列号和元素值称为该元素的三元组。

在 Python 语言中稀疏矩阵的三元组表示的节点结构定义如下：

```
class TripleNode(object):
    def __init__(self,row = 0,column = 0,value = 0):
        self.row = row
        self.column = column
        self.value = value
```

稀疏矩阵的三元组顺序表类的定义如下：

```
class SparseMatrix(object):
    def __init__(self,maxSize):
        self.maxSize = maxSize
        self.data = [None] * self.maxSize     # 三元组表
        for i in range(self.maxSize):
            self.data[i] = TripleNode()
        self.rows = 0                         # 行数
        self.cols = 0                         # 列数
        self.nums = 0                         # 非零元素个数
```

初始化三元组顺序表是按先行序后列序的原则扫描稀疏矩阵，并把非零元素插入到顺

序表中,其算法如下。

【算法 4.8】 初始化三元组顺序表。

```
def create(self,mat):
    count = 0
    self.rows = len(mat)
    self.cols = len(mat[0])
    for i in range(self.rows):
        for j in range(self.cols):
            if mat[i][j]!= 0:
                count += 1
    self.num = count
    self.data = [None] * self.nums
    for i in range(self.rows):
        for j in range(self.cols):
            if mat[i][j]!= 0:
                self.data[k] = TripleNode(i,j,mat[i][j])
                k += 1
```

2. 稀疏矩阵的十字链表存储

当稀疏矩阵中非零元素的位置或个数经常发生变化时不宜采用三元组顺序表存储结构,而应该采用链式存储结构表示。十字链表是稀疏矩阵的另一种存储结构,在十字链表中稀疏矩阵的非零元素用一个节点来表示,每个节点由 5 个域组成。row 域存放该元素的行号,column 域存放该元素的列号,value 域存放该元素的值,right 域存放与该元素同行的下一个非零元素节点的指针,down 域存放与该元素同列的下一个非零元素节点的指针。每个非零数据元素节点既是某个行链表中的一个节点,也是某个列链表中的节点,整个稀疏矩阵构成了一个十字交叉的链表,这样的链表就称为十字链表。

在 Python 语言中可以将稀疏矩阵的十字链表表示的节点结构定义如下:

```
class OLNode(object):
    def __init__(self,row = 0,col = 0,value = 0):
        self.row = row          # 行号
        self.col = col          # 列号
        self.value = value      # 数据元素值
        self.right = None       # 行链表指针
        self.down = None        # 列链表指针
```

稀疏矩阵的十字链表类的定义如下:

```
class CrossList(object):
    def __init__(self,rows,cols):
        self.rows = rows        # 十字链表的行数
        self.cols = cols        # 十字链表列数
        self.nums = 0           # 非零元素的个数
         # 行列的指针数组
        self.rhead = [None] * rows
        self.chead = [None] * cols
```

【例 4.5】 已知 **A** 为稀疏矩阵,试从空间和时间角度比较采用二维数组和三元组顺序

表两种不同的存储结构完成求运算的优缺点。

解:

设稀疏矩阵为 m 行 n 列,如果采用二维数组存储,其空间复杂度为 $O(m\times n)$;因为要将所有的矩阵元素累加起来,所以需要用一个两层的嵌套循环,其时间复杂度也为 $O(m\times n)$。

如果采用三元组顺序表进行压缩存储,假设矩阵中有 t 个非零元素,其空间复杂度为 $O(t)$,将所有的矩阵元素累加起来只需将三元组顺序表扫描一遍,其时间复杂度也为 $O(t)$。当 $t\ll m\times n$ 时采用三元组顺序表存储可获得较好的时空性能。

小　　结

(1) 字符串是数据元素类型为字符的线性表,串具有插入、删除、链接、查找、比较等基本操作。

(2) 字符串具有顺序存储结构和链式存储结构两种存储结构。字符串的顺序存储结构叫顺序串,与顺序表的逻辑结构相同,存储结构类似,均可用数组来存储数据元素。字符串的链式存储结构叫链串,和线性表的链式存储结构类似,可以采用单链表存储串值。链串由一系列大小相同的节点组成,每个节点用数据域存放字符,指针域存放指向下一个节点的指针。

(3) 串的模式匹配也叫查找定位,指的是在当前串中寻找模式串的过程,主要的模式匹配算法有 Brute Force 算法和 KMP 算法。

(4) 数组是 n 个具有相同数据类型的数据元素构成的集合,数组元素按某种次序存储在地址连续的存储单元中,是一种随机存储结构。

(5) 特殊矩阵是具有许多相同数据元素或者零元素且数据元素的分布具有一定规律的矩阵,例如对称矩阵、三角矩阵和对角矩阵。为了节省存储空间,对矩阵进行压缩存储。特殊矩阵的压缩存储方法是将呈现规律性分布的、值相同的多个矩阵元素压缩存储到一个存储空间。

(6) 稀疏矩阵是具有较多零元素,并且非零元素的分布无规律的矩阵。稀疏矩阵的压缩存储是只给非零数据元素分配存储空间。

习　题　4

一、选择题

1. 串是一种特殊的线性表,其特殊性体现在(　　)。

A. 可以顺序存储　　B. 数据元素是一个字符

C. 可以链式存储　　D. 数据元素可以是多个字

2. 设有两个串 p 和 q,求 q 在 p 中首次出现的位置的运算称为(　　)。

A. 连接　　B. 模式匹配　　C. 求子串　　D. 求串长

3. 设串 $s1$="ABCDEFG"、$s2$="PQRST",函数 con(x,y)返回 x 和 y 串的连接串,subs(s,i,j)返回串 s 的从序号 i 开始的 j 个字符组成的子串,len(s)返回串 s 的长度,则

con(subs($s1$,2,len($s2$)),subs($s1$,len($s2$),2))的结果串是(　　)。

A. BCDEF　　B. BCDEFG　　C. BCPQRST　　D. BCDEFEF

4. 假设有60行70列的二维数组a[1..60, 1..70]以列序为主序顺序存储,其基地址为10000,每个元素占两个存储单元,那么第32行第58列的元素a[32,58]的存储地址为(　　),注意无第0行第0列元素。

A. 16902　　B. 16904

C. 14454　　D. 答案A、B、C均不对

5. 设矩阵**A**是一个对称矩阵,为了节省存储,将其下三角部分(如图4.5所示)按行序存放在一维数组B[1, $n(n-1)/2$]中,

$$\mathbf{A}=\begin{bmatrix} a_{1,1} & & \\ \vdots & \ddots & \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

图4.5　矩阵**A**的下三角部分

对下三角部分中的任一元素$a_{i,j}$($i \leqslant j$),在一维数组B中下标k值是(　　)。

A. $i(i-1)/2+j-1$　　B. $i(i-1)/2+j$

C. $i(i+1)/2+j-1$　　D. $i(i+1)/2+j$

6. 从供选择的答案中选出应填入下面叙述中的最确切的解答,把相应编号写在答卷的对应栏内。

有一个二维数组A,行下标的范围是0到8,列下标的范围是1到5,每个数组元素用相邻的4个字节存储,存储器按字节编址。假设存储数组元素A[0,1]的第一个字节的地址是0,存储数组A的最后一个元素的第一个字节的地址是①。若按行存储,则A[3,5]和A[5,3]的第一个字节的地址分别是②和③。若按列存储,则A[7,1]和A[2,4]的第一个字节的地址分别是④和⑤。

供选择的答案:

①～⑤:A. 28　　B. 44　　C. 76　　D. 92

E. 108　　F. 116　　G. 132　　H. 176

I. 184　　J. 188

7. 有一个二维数组A,行下标的范围是1到6,列下标的范围是0到7,每个数组元素用相邻的6个字节存储,存储器按字节编址。那么,这个数组的体积是①个字节。假设存储数组元素A[1,0]的第一个字节的地址是0,则存储数组A的最后一个元素的第一个字节的地址是②。若按行存储,则A[2,4]的第一个字节的地址是③。若按列存储,则A[5,7]的第一个字节的地址是④。

供选择的答案:

①～④:A. 12　　B. 66　　C. 72　　D. 96

E. 114　　F. 120　　G. 156　　H. 234

I. 276　　J. 282　　K. 283　　L. 288

二、填空题

1. 不包含任何字符(长度为0)的串称为________;由一个或多个空格(仅有空格符)组

成的串称为________。

2. 设 s="A:/document/Mary.doc",则 strlen(s)= ________的"/"字符定位的位置为________。

3. 子串的定位运算称为串的模式匹配；被匹配的主串称为________，________称为模式。

4. 三元组表中的每个节点对应于稀疏矩阵的一个非零元素,它包含有 3 个数据项,分别表示该元素的________、________、________。

5. 设目标 T="abccdcdccbaa",模式 P="cdcc",则第________次匹配成功。

6. 若 n 为主串长、m 为子串长,则串的古典(朴素)匹配算法在最坏情况下需要比较字符的总次数为________。

7. 假设有二维数组 A6×8,每个元素用相邻的 6 个字节存储,存储器按字节编址。已知 A 的起始存储位置(基地址)为 1000,则数组 A 的体积(存储量)为________；末尾元素 $A57$ 的第一个字节的地址为________；若按行存储,元素 $A14$ 的第一个字节的地址为________；若按列存储,元素 $A47$ 的第一个字节的地址为________。

8. 设数组 a[1..60,1..70]的基地址为 2048,每个元素占两个存储单元,若以列序为主序顺序存储,则元素 a[32,58]的存储地址为________。

三、算法设计题

1. 若在矩阵 **A** 中存在一个元素 $a_{i,j}$($0\leqslant i\leqslant n-1,0\leqslant j\leqslant m-1$),该元素是第 i 行元素中的最小值且又是第 j 列元素中的最大值,则称此元素为该矩阵的一个马鞍点。假设以二维数组存储矩阵 **A**,试设计一个求该矩阵的所有马鞍点的算法,并分析最坏情况下的时间复杂度。

2. 编写基于 SeqString 类的成员函数 count(),统计当前字符串中的单词个数。

3. 编写基于 SeqString 类的成员函数 reverse(),要求将当前对象中的字符反序存放。

4. 编写基于 SeqString 类的成员函数 deleteallchar(ch),要求从当前对象串中删除其值等于 ch 的所有字符。

5. 编写基于 SeqString 类的成员函数 stringcount(str),要求统计子串 str 在当前对象串中出现的次数,若不出现则返回 0。

6. 在顺序串类 SeqString 中增加一个主函数,测试各成员函数的正确性。

7. 已知两个稀疏矩阵 ***A*** 和 ***B***,试基于三元组顺序表或十字链表的存储链表编程实现 ***A***+***B*** 的运算。

第5章 树结构

5.1 树

5.1.1 树的基本概念

树是数据元素之间具有层次关系的非线性结构，是由 n 个节点构成的有限集合，节点数为 0 的树叫空树。树必须满足以下条件。

(1) 有且仅有一个被称为根的节点。

(2) 其余节点可分为 m 个互不相交的有限集合，每个集合又构成一棵树，叫根节点的子树。

与线性结构不同，树中的数据元素具有一对多的逻辑关系，除根节点以外，每个数据元素可以有多个后继但有且仅有一个前驱，反映了数据元素之间的层次关系。

树是递归定义的。节点是树的基本单位，若干个节点组成一棵子树，若干棵互不相交的子树组成一棵树。

人们在生活中所见的家谱、Windows 的文件系统等，虽然表现形式各异，但在本质上是树结构。图 5.1 给出了树的逻辑结构示意图。

树的表示方法有多种，如树形表示法、文氏图表示法、凹入图表示法和广义表表示法。图 5.1 所示为树形表示法，图 5.2 给出了用其他 3 种表示法对树的表示。

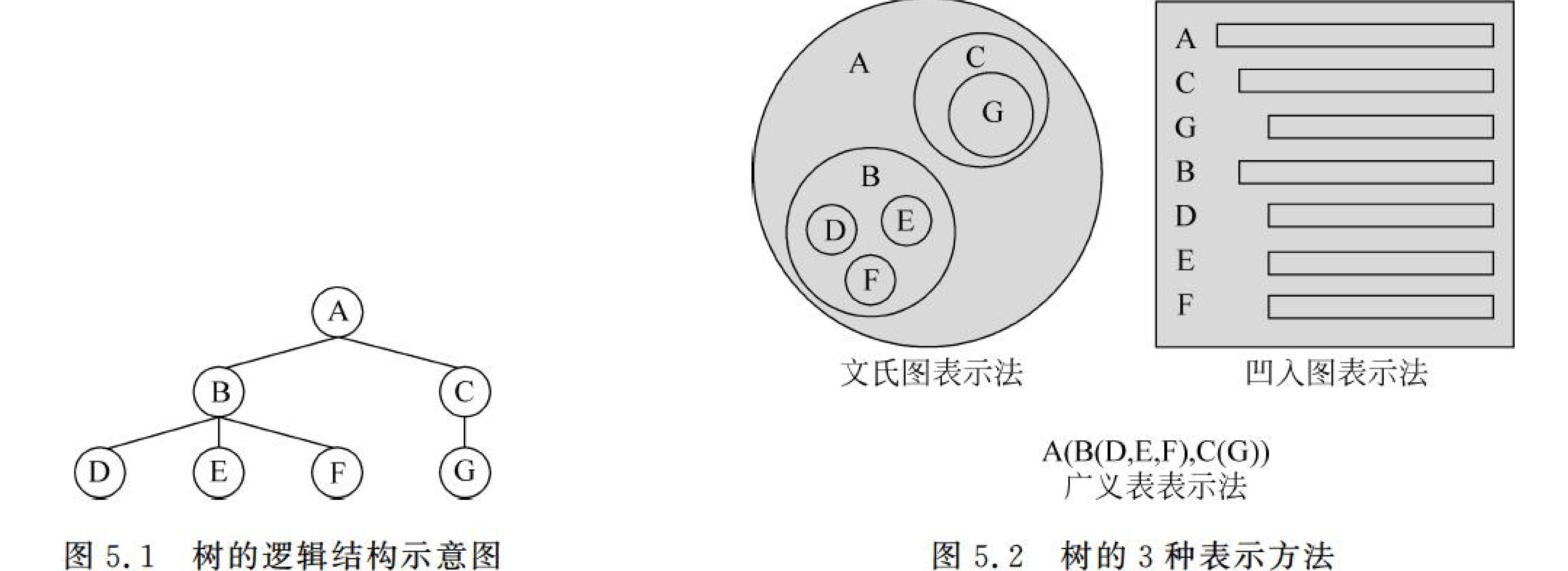


图 5.1 树的逻辑结构示意图

图 5.2 树的 3 种表示方法

5.1.2 树的术语

1. 节点

树的节点就是构成树的数据元素，就是其他数据结构中存储的数据项，在树形表示法中

用圆圈表示。

2. 节点的路径

节点的路径是指从根节点到该节点所经过节点的顺序排列。

3. 路径的长度

路径的长度指的是路径中包含的分支数。

4. 节点的度

节点的度指的是节点拥有的子树的数目。

5. 树的度

树的度指的是树中所有节点的度的最大值。

6. 叶节点

叶节点是树中度为 0 的节点,也叫终端节点。

7. 分支节点

分支节点是树中度不为 0 的节点,也叫非终端节点。

8. 子节点

子节点是指节点的子树的根节点,也叫孩子节点。

9. 父节点

具有子节点的节点叫该子节点的父节点,也叫双亲节点。

10. 子孙节点

子孙节点是指节点的子树中的任意节点。

11. 祖先节点

祖先节点是指节点的路径中除自身之外的所有节点。

12. 兄弟节点

兄弟节点是指和节点具有同一父节点的节点。

13. 节点的层次

树中根节点的层次为 0,其他节点的层次是父节点的层次加 1。

14. 树的深度

树的深度是指树中所有节点的层次数的最大值加 1。

15. 有序树

有序树是指树的各节点的所有子树具有次序关系,不可以改变位置。

16. 无序树

无序树是指树的各节点的所有子树之间无次序关系,可以改变位置。

17. 森林

森林是由多个互不相交的树构成的集合。给森林加上一个根节点就变成一棵树,将树的根节点删除就变成森林。

5.2 二 叉 树

5.2.1 二叉树的基本概念

1. 普通二叉树

二叉树是特殊的有序树,它也是由 n 个节点构成的有限集合。当 $n=0$ 时称为空二叉

树。二叉树的每个节点最多只有两棵子树，子树也为二叉树，互不相交且有左右之分，分别称为左二叉树和右二叉树。

二叉树也是递归定义的，在树中定义的度、层次等术语同样也适用于二叉树。

2. 满二叉树

满二叉树是特殊的二叉树，它要求除叶节点外的其他节点都具有两棵子树，并且所有的叶节点都在同一层上，如图 5.3 所示。

3. 完全二叉树

完全二叉树是特殊的二叉树，若完全二叉树具有 n 个节点，它要求 n 个节点与满二叉树的前 n 个节点具有完全相同的逻辑结构，如图 5.4 所示。

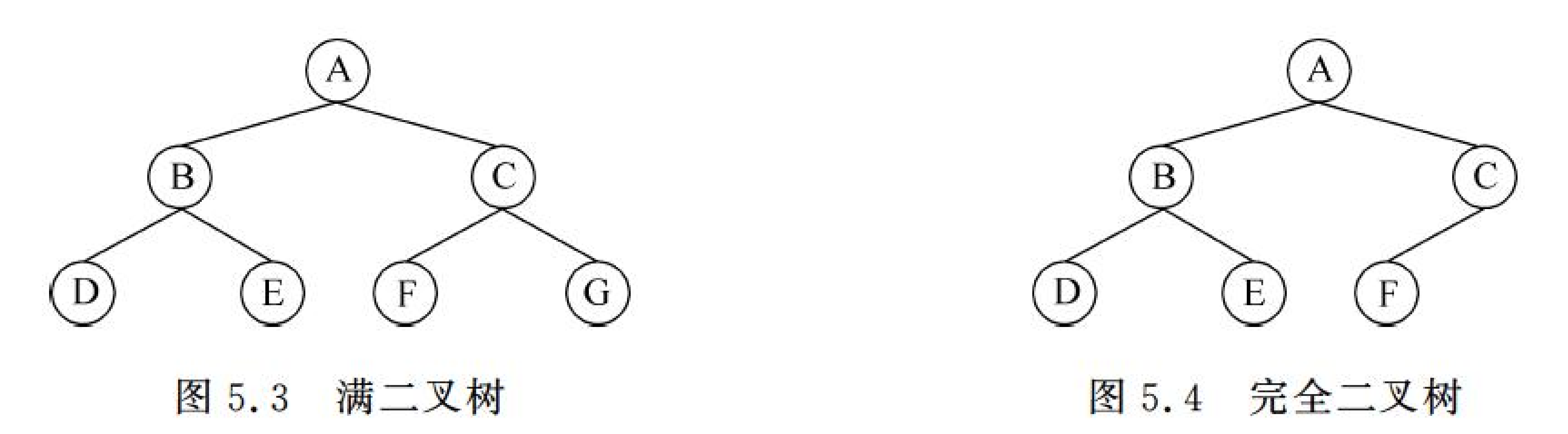


图 5.3　满二叉树　　　　图 5.4　完全二叉树

5.2.2　二叉树的性质

性质 1：二叉树中第 i 层的节点数最多为 $2i$。

证明：当 $i=0$ 时只有一个根节点，成立；假设对所有的 $k(0\leqslant k<i)$ 成立，即第 $i-1$ 层上最多有 2^{i-1} 个节点，那么由于每个节点最多有两棵子树，在第 i 层上节点数最多为 $2^{i-1}\times 2=2^i$ 个，得证。

性质 2：深度为 h 的二叉树最多有 2^h-1 个节点。

证明：由性质 1 得，深度为 h 的二叉树的节点个数最多为 $2^0+2^1+\cdots+2^{h-1}=2^h-1$，得证。

性质 3：若二叉树的叶节点的个数为 n，度为 2 的节点个数为 m，有 $n=m+1$。

证明：设二叉树中度为 1 的节点个数为 k，二叉树的节点总数为 s，有 $s=k+n+m$。又因为除根节点外每个节点都有一个进入它的分支，所以 $s-1=k+2\times m$。整理后得到 $n=m+1$，得证。

性质 4：具有 n 个节点的完全二叉树，其深度为 $\lfloor \log_2 n \rfloor+1$ 或者 $\lceil \log_2(n+1) \rceil$。

证明：设此二叉树的深度为 h，由性质 2 可得 $2^{h-1}\leqslant n<2^h$，两边取对数，可得 $h-1\leqslant \log_2 n<h$，因为 h 为整数，所以 $h=\lfloor \log_2 n \rfloor+1$，得证。

性质 5：具有 n 个节点的完全二叉树，从根节点开始自上而下、从左向右对节点从 0 开始编号。对于任意一个编号为 i 的节点：

(1) 若 $i=0$，节点为根节点，没有父节点；若 $i>0$，则父节点的编号为 $\lfloor (i-1)/2 \rfloor$。

(2) 若 $2i+1\geqslant n$，该节点无左孩子，否则左孩子节点的编号为 $2i+1$。

(3) 若 $2i+2\geqslant n$，该节点无右孩子，否则右孩子节点的编号为 $2i+2$。

【例 5.1】　对于任意一个满二叉树，其分支数 $B=2(n_0-1)$，其中 n_0 为终端节点数。

解：

设 n_2 为度为 2 的节点，因为在满二叉树中没有度为 1 的节点，所以有：

$$n = n_0 + n_2$$

设 B 为树中分支数，则：

$$n = B + 1$$

所以：

$$B = n_0 + n_2 - 1$$

再由二叉树的性质：

$$n_0 = n_2 + 1$$

代入上式有：

$$B = n_0 + n_0 - 1 - 1 = 2(n_0 - 1)$$

【例 5.2】 已知一棵度为 m 的树中有 n_1 个度为 1 的节点、n_2 个度为 2 的节点、…、n_m 个度为 m 的节点，问该树中共有多少个叶子节点？

解：设该树的总节点数为 n，则：

$$n = n_0 + n_1 + n_2 + \cdots + n_m$$

又

$$n = \text{分支数} + 1 = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 + \cdots + m \times n_m + 1$$

由上述两式可得：

$$n_0 = n_2 + 2n_3 + \cdots + (m-1)n_m + 1$$

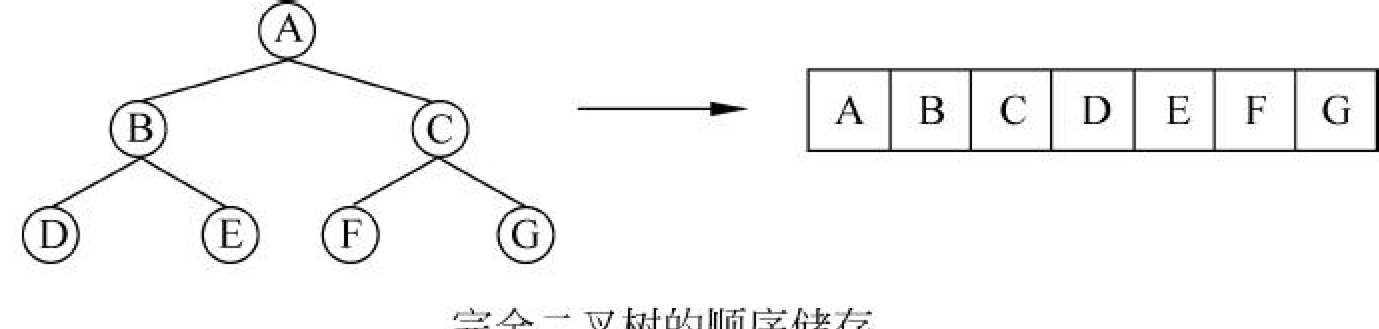
5.2.3 二叉树的存储结构

1. 二叉树的顺序存储结构

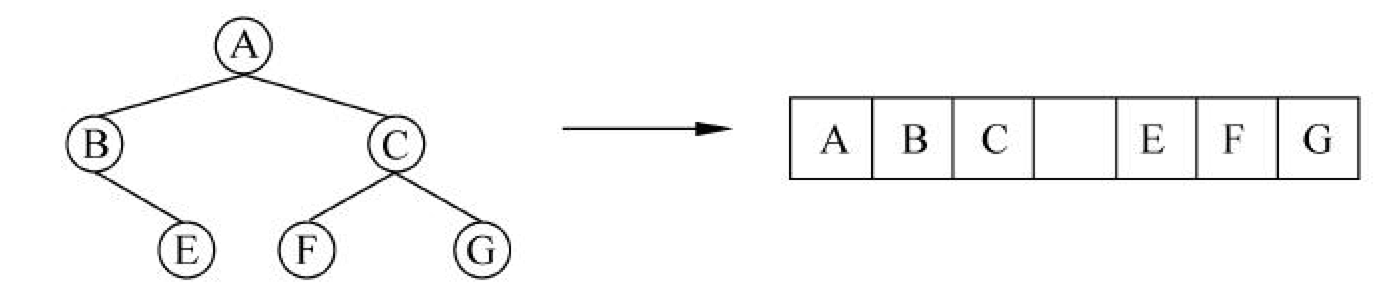
二叉树的顺序存储结构是指将二叉树的各个节点存放在一组地址连续的存储单元中，所有节点按节点序号进行顺序存储。因为二叉树为非线性结构，所以必须先将二叉树的节点排成线性序列再进行存储，实际上是对二叉树先进行一次层次遍历。二叉树的各节点间的逻辑关系由节点在线性序列中的相对位置确定。

可以利用 5.2.2 节中的性质 5 将二叉树的节点排成线性序列，将节点存放在下标为对应编号的数组元素中。为了存储非完全二叉树，需要在树中添加虚节点使其成为完全二叉树后再进行存储，这样会造成存储空间的浪费。

图 5.5 所示为二叉树的顺序存储结构示意图。



完全二叉树的顺序储存



非完全二叉树的顺序储存

图 5.5 二叉树的顺序存储结构

2. 二叉树的链式存储结构

二叉树的链式存储结构是指将二叉树的各个节点随机存放在存储空间中,二叉树的各节点间的逻辑关系由指针确定。每个节点至少要有两条链分别连接左、右孩子节点才能表达二叉树的层次关系。

根据指针域个数的不同,二叉树的链式存储结构又分为以下两种。

1) 二叉链式存储结构

二叉树的每个节点设置两个指针域和一个数据域。数据域中存放节点的值,指针域中存放左、右孩子节点的存储地址。

采用二叉链表存储二叉树,每个节点只存储了到其孩子节点的单向关系,没有存储到其父节点的关系,因此要获得父节点将花费较多的时间,需要从根节点开始在二叉树中进行查找,所花费的时间是遍历部分二叉树的时间,且与查找节点所处的位置有关。

2) 三叉链式存储结构

二叉树的每个节点设置 3 个指针域和一个数据域。数据域中存放节点的值,指针域中存放左、右孩子节点和父节点的存储地址。

图 5.6 所示为二叉链式存储和三叉链式存储的节点结构。

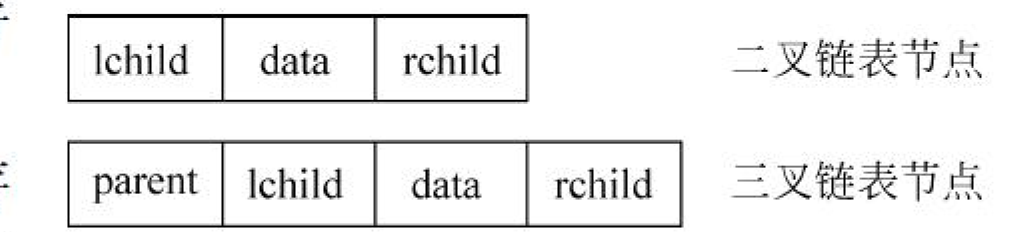


图 5.6 二叉和三叉链式存储的节点结构

两种链式存储结构各有优缺点,二叉链式存储结构空间利用率高,而三叉链式存储结构既便于查找孩子节点,又便于查找父节点。在实际应用中,二叉链式存储结构更加常用,因此本书中二叉树的相关算法都是基于二叉链式存储结构设计的。

3. 二叉链式存储结构的节点类的描述

```
class BiTreeNode(metaclass = ABCMeta):
    def __init__(self,data = None,lchild = None,rchild = None):
        self.data = data          # 数据域的值
        self.lchild = lchild      # 左孩子的指针
        self.rchild = rchild      # 右孩子的指针
```

4. 二叉树类的描述

此二叉树类基于二叉链式存储结构实现。

```
class BiTree(object):
    def __init__(self,root = None):
        self.root = root # 二叉树的根节点
```

二叉树的创建操作和遍历操作比较重要,将在下面的章节中进行详细介绍。

5.2.4 二叉树的遍历

二叉树的遍历是指沿着某条搜索路径访问二叉树的节点,每个节点被访问的次数有且仅有一次。

1. 二叉树的遍历方法

二叉树通常可划分为 3 个部分,即根节点、左子树和右子树。根据 3 个部分的访问顺序

不同,可将二叉树的遍历方法分为以下几种。

1) 层次遍历

自上而下、从左到右依次访问每层的节点。

2) 先序遍历

先访问根节点,再先序遍历左子树,最后先序遍历右子树。

3) 中序遍历

先中序遍历左子树,再访问根节点,最后中序遍历右子树。

4) 后序遍历

先后序遍历左子树,再后序遍历右子树,最后访问根节点。

图 5.7 描述了先序遍历、中序遍历和后序遍历序列的节点排列规律。

先序遍历 根 左子树 右子树
中序遍历 左子树 根 右子树
后序遍历 左子树 右子树 根

图 5.7 二叉树遍历序列的节点排列规律

2. 二叉树遍历操作实现的递归算法

【算法 5.1】 先序遍历。

```
def preOrder(root):
    if root is not None:
        print(root.data,end = ' ')
        BiTree.preOrder(root.lchild)
        BiTree.preOrder(root.rchild)
```

【算法 5.2】 中序遍历。

```
def inOrder(root):
    if root is not None:
        BiTree.inOrder(root.lchild)
        print(root.data,end = ' ')
        BiTree.inOrder(root.rchild)
```

【算法 5.3】 后序遍历。

```
def postOrder(root):
    if root is not None:
        BiTree.postOrder(root.lchild)
        BiTree.postOrder(root.rchild)
        print(root.data,end = ' ')
```

3. 二叉树遍历操作实现的非递归算法

二叉树遍历操作的递归算法结构简洁,易于实现,但是在时间上开销较大,运行效率较低,为了解决这一问题,可以将递归算法转换为非递归算法,转换方式有以下两种。

(1) 使用临时遍历保存中间结果,用循环结构代替递归过程。

(2) 利用栈保存中间结果。

二叉树遍历操作实现的非递归算法利用栈结构通过回溯访问二叉树的每个节点。

1) 先序遍历

先序遍历从二叉树的根节点出发,沿着该节点的左子树向下搜索,每遇到一个节点先访

问该节点,并将该节点的右子树入栈。先序遍历左子树完成后再从栈顶弹出右子树的根节点,然后采用相同的方法先序遍历右子树,直到二叉树的所有节点都被访问。其主要步骤如下。

(1) 将二叉树的根节点入栈。

(2) 若栈非空,将节点从栈中弹出并访问。

(3) 依次访问当前访问节点的左孩子节点,并将当前节点的右孩子节点入栈。

(4) 重复步骤(2)和(3),直到栈为空。

【算法 5.4】 先序遍历。

```
def preOrder2(root):
    p = root
    s = LinkStack()
    s.push(p)
    while not s.isEmpty():
        p = s.pop()
        print(p.data,end = ' ')
        while p is not None:
            if p.lchild is not None:
                print(p.lchild.data,end = ' ')
            if p.rchild is not None:
                s.push(p.rchild)
            p = p.lchild
```

2) 中序遍历

中序遍历从二叉树的根节点出发,沿着该节点的左子树向下搜索,每遇到一个节点就使其入栈,直到节点的左孩子节点为空。再从栈顶弹出节点并访问,然后采用相同的方法中序遍历节点的右子树,直到二叉树的所有节点都被访问。其主要步骤如下。

(1) 将二叉树的根节点入栈。

(2) 若栈非空,将栈顶节点的左孩子节点依次入栈,直到栈顶节点的左孩子节点为空。

(3) 将栈顶节点弹出并访问,并使栈顶节点的右孩子节点入栈。

(4) 重复步骤(2)和(3),直到栈为空。

【算法 5.5】 中序遍历。

```
def inOrder2(root):
    p = root
    s = LinkStack()
    s.push(p)
    while not s.isEmpty():
        while p.lchild is not None:
            p = p.lchild
            s.push(p)
        p = s.pop()
        print(p.data,end = ' ')
        if p.rchild is not None:
            s.push(p.rchild)
```

3）后序遍历

后序遍历从二叉树的根节点出发，沿着该节点的左子树向下搜索，每遇到一个节点需要判断其是否为第一次经过，若是则使节点入栈，后序遍历该节点的左子树，完成后再遍历该节点的右子树，最后从栈顶弹出该节点并访问。后序遍历算法的实现需要引入两个变量，一个为访问标记变量 flag，用于标记栈顶节点是否被访问，若 flag＝true，证明该节点已被访问，其左子树和右子树已经遍历完毕，可继续弹出栈顶节点，否则需要先遍历栈顶节点的右子树；一个为节点指针 t，指向最后一个被访问的节点，查看栈顶节点的右孩子节点，证明此节点的右子树已经遍历完毕，栈顶节点可出栈并访问。其主要步骤如下。

（1）将二叉树的根节点入栈，t 赋值为空。

（2）若栈非空，将栈顶节点的左孩子节点依次入栈，直到栈顶节点的左孩子节点为空。

（3）若栈非空，查看栈顶节点的右孩子节点，若右孩子节点为空或者与 p 相等，则弹出栈顶节点并访问，同时使 t 指向该节点，并置 flag 为 true；否则将栈顶节点的右孩子节点入栈，并置 flag 为 false。

（4）若 flag 为 true，重复步骤（3）；否则重复步骤（2）和（3），直到栈为空。

【算法 5.6】 后序遍历。

```
def postOrder2(root):
    p = root
    t = None
    flag = True
    s = LinkStack()
    if p is not None:
        s.push(p)
        while p.child is None:
            p = p.lchild
            s.push(p)
        while not s.isEmpty() and flag:
            if p.rchild == t or p.rchild is None:
                print(p.data,end='')
                flag = True
                t = p
                s.pop()
            else:
                s.push(p.rchild)
                flag = False
```

4）层次遍历

层次遍历操作是从根节点出发，自上而下、从左到右依次遍历每层的节点，可以利用队列先进先出的特性进行实现。先将根节点入队，然后将队首节点出队并访问，都将其孩子节点依次入队。其主要步骤如下。

（1）将根节点入队。

（2）若队非空，取出队首节点并访问，将队首节点的孩子节点入队。

（3）重复执行步骤（2）直到队为空。

【算法 5.7】 层次遍历。

```
def order(root):
    q = LinkQueue()
    q.offer(root)
    while not q.isEmpty():
        p = q.poll()
        print(p.data,end = '')
        if p.lchild is not None:
            q.offer(p.lchild)
        if p.rchild is not None:
            q.offer(p.rchild)
```

对于有 n 个节点的二叉树,因为每个节点都只访问一次,所以以上 4 种遍历算法的时间复杂度均为 $O(n)$。

4 种遍历算法的实现均利用了栈或队列,增加了额外的存储空间,存储空间的大小为遍历过程中栈或队列需要的最大容量。对于栈来说,其最大容量即为树的高度,在最坏情况下有 n 个节点的二叉树的高度为 n,所以其空间复杂度为 $O(n)$;对于队列来说,其最大容量为二叉树相邻两层的最大节点总数,与 n 成线性关系,所以其空间复杂度也为 $O(n)$。

5.2.5 二叉树遍历算法的应用

二叉树的遍历操作是实现对二叉树其他操作的一个重要基础,本节介绍了二叉树遍历算法在许多应用问题中的运用。

1. 二叉树上的查找算法

二叉树上的查找是在二叉树中查找值为 x 的节点,若找到返回该节点,否则返回空值,可以在二叉树的先序遍历过程中进行查找,主要步骤如下。

(1) 若二叉树为空,则不存在值为 x 的节点,返回空值;否则将根节点的值与 x 进行比较,若相等,返回该节点。

(2) 若根节点的值与 x 的值不等,则在左子树中进行查找,若找到,则返回该节点。

(3) 若没有找到,则在根节点的右子树中进行查找,若找到,返回该节点,否则返回空值。

【算法 5.8】 二叉树查找算法。

```
def searchNode(t,x):
    if t is None:
        return None
    if t.data == x:
        return t
    else:
        lresult = searchNode(t.lchild,x)
    if lresult == None:
        return searchNode(t.rchild,x)
    else:
        return lresult
```

2. 统计二叉树的节点个数的算法

二叉树的节点个数等于根节点加上左、右子树的节点的个数,可以利用二叉树的先序遍历序列,引入一个计数变量 count,count 的初值为 0,每访问根节点一次就将 count 的值加 1,其主要操作步骤如下。

(1) count 值初始化为 0。

(2) 若二叉树为空,返回 count 值。

(3) 若二叉树非空,则 count 值加 1,统计根节点的左子树的节点个数,并将其加到 count 中;统计根节点的右子树的节点个数,并将其加到 count 中。

【算法 5.9】 统计二叉树的节点个数。

```
def nodeCount(t):
    count = 0
    if t is not None:
        count += 1
        count += nodeCount(t.lchild)
        count += nodeCount(t.rchild)
    return count
```

3. 求二叉树的深度

二叉树的深度是所有节点的层次数的最大值加 1,也就是左子树和右子树的深度的最大值加 1,可以采用后序遍历的递归算法解决此问题,其主要步骤如下。

(1) 若二叉树为空,返回 0。

(2) 若二叉树非空,求左子树的深度,求右子树的深度。

(3) 比较左、右子树的深度,取最大值加 1 即为二叉树的深度。

【算法 5.10】 求二叉树的深度。

```
def getDepth(t):
    if t is None:
        return 0
    ldepth = getDepth(t.lchild)
    rdepth = getDepth(t.rchild)
    if ldepth < rdepth:
        return rdepth + 1
    else:
        return ldepth + 1
```

5.2.6 二叉树的建立

二叉树遍历操作可使非线性结构的树转换成线性序列。先序遍历序列和后序遍历序列反映父节点和孩子节点间的层次关系,中序遍历序列反映兄弟节点间的左右次序关系。因为二叉树是具有层次关系的节点构成的非线性结构,并且每个节点的孩子节点具有左右次序,所以已知一种遍历序列无法唯一确定一棵二叉树,只有同时知道中序和先序遍历序列,或者同时知道中序和后序遍历序列,才能同时确定节点的层次关系和节点的左右次序,才能唯一确定一棵二叉树。

1. 由中序和先序遍历序列建立二叉树

其主要步骤为如下。

(1) 取先序遍历序列的第一个节点作为根节点,序列的节点个数为 n。

(2) 在中序遍历序列中寻找根节点,其位置为 i,可确定在中序遍历序列中根节点之前的 i 个节点构成的序列为根节点的左子树中序遍历序列,根节点之后的 $n-i-1$ 个节点构成的序列为根节点的右子树中序遍历序列。

(3) 在先序遍历序列中根节点之后的 i 个节点构成的序列为根节点的左子树先序遍历序列,先序遍历序列之后的 $n-i-1$ 个节点构成的序列为根节点的右子树先序遍历序列。

(4) 对左、右子树重复步骤(1)、(2)、(3),确定左、右子树的根节点和子树的左右、子树。

(5) 算法递归进行即可建立一棵二叉树。

假设二叉树的先序遍历序列为 ABECFG,中序遍历序列为 BEAFCG,由中序和先序遍历序列建立二叉树的过程如图 5.8 所示。

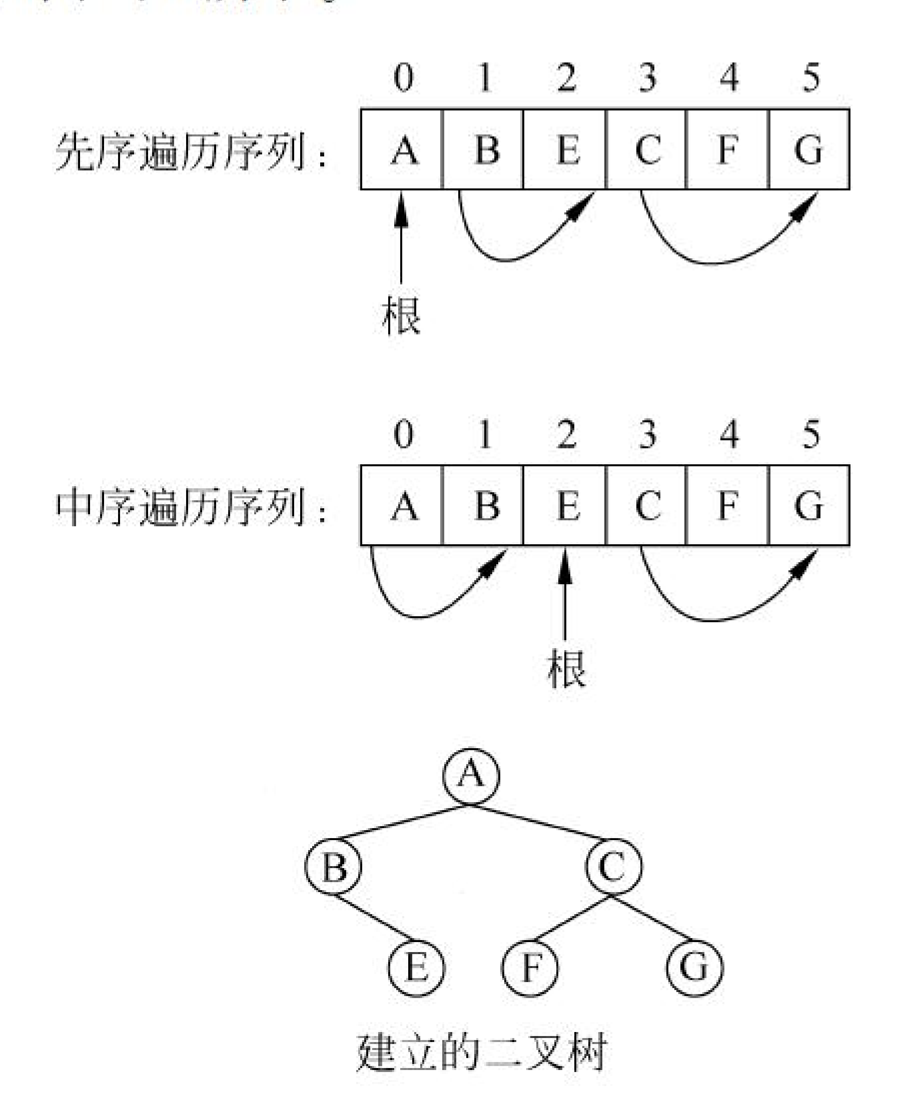


图 5.8 由中序和先序遍历序列建立二叉树

【算法 5.11】 由中序和先序遍历序列建立二叉树。

```
def createBiTree(preOrder,inOrder,preo,ino,n):
    if n > 0:
        i = 0
        c = preOrder.charAt(preo) # c 为先序序列的根节点
        while i < n:
            if inOrder.charAt(i + ino) == c:
                break
            i += 1
        root = BiTreeNode(c)
        root.lchild = createBiTree(preOrder,inOrder,preo + 1,ino,i).root
        # 递归寻找左子树的根节点
        root.rchild = createBiTree(preOrder,inOrder,preo + i + 1,ino + i + 1,n - i - 1).root
        # 递归寻找右子树的根节点
```

2. 由标明空子树的先序遍历序列创建二叉树

其主要步骤如下。

(1) 从先序遍历序列中依次读取字符。

(2) 若字符为#,建立空子树。

(3) 建立左子树。

(4) 建立右子树。

【算法 5.12】 由标明空子树的先序遍历序列建立二叉树。

```
def createBiTree(preOrder,i): # i 为常数 0
    c = preOrder.charAt(i) # 取字符
    if c != '#':
        root = BiTreeNode(c)
        i += 1
        root.lchild = createBiTree(preOrder,i).root
        i += 1
        root.rchild = createBiTree(preOrder,i).root
    else:
        root = None
```

【例 5.3】 已知二叉树的中序和后序序列分别为 CBEDAFIGH 和 CEDBIFHGA,试构造该二叉树。

解:二叉树的构造过程如图 5.9 所示。图(c)即为构造出的二叉树。

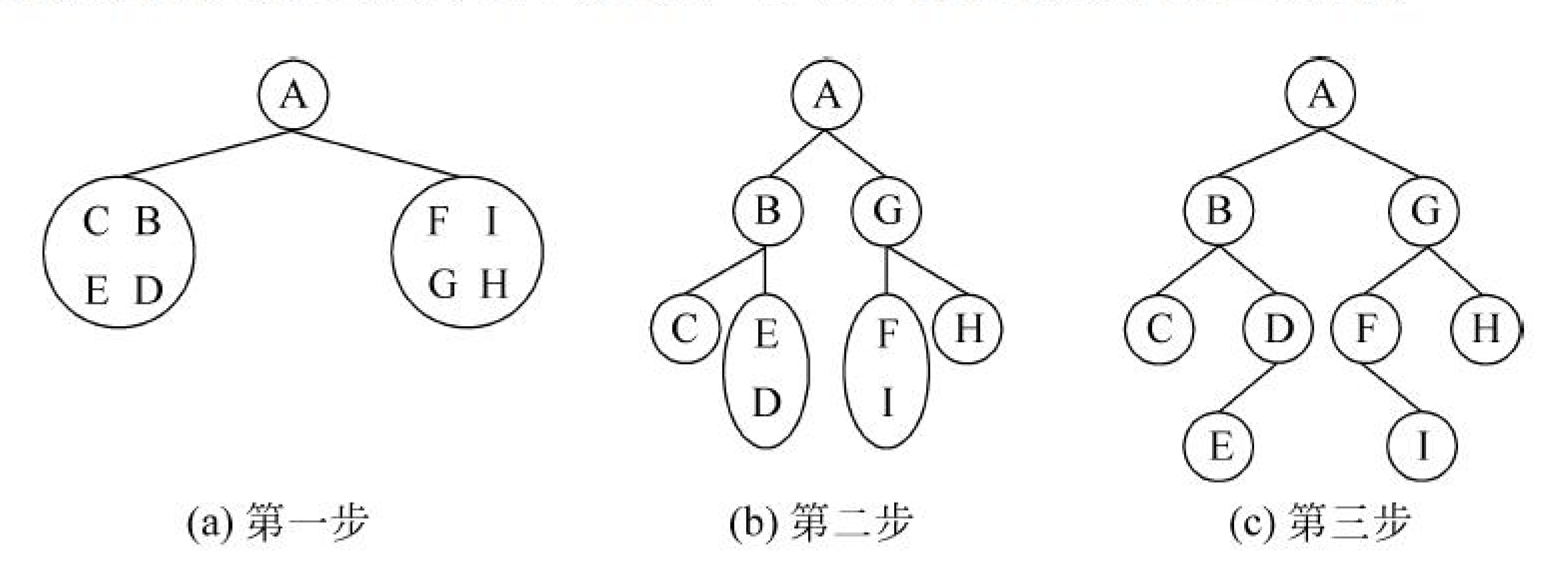


图 5.9 二叉树的构造过程

5.3 哈夫曼树及哈夫曼编码

目前常用的图像、音频、视频等多媒体信息数据量大,必须对它们采用数据压缩技术来存储和传输。数据压缩技术通过对数据进行重新编码来压缩存储,以便减少数据占用的存储空间,在使用时再进行解压缩,恢复数据的原有特性。

其压缩方法主要有有损压缩和无损压缩两种。有损压缩是指压缩过程中可能会丢失数据信息,如将 BMP 位图压缩成 JPEG 格式的图像,会有精度损失;无损压缩是指压缩存储数据的全部信息,确保解压后的数据不丢失。哈夫曼编码是数据压缩技术中的无损压缩技术。

5.3.1 哈夫曼树的基本概念

1. 节点间的路径

节点间的路径是指从一个节点到另一个节点所经过的节点序列。从根节点到 X 节点有且仅有一条路径。

2. 节点的路径长度

节点的路径长度是指从根节点到节点的路径上的边数。

3. 节点的权

节点的权是指人给节点赋予的一个具有某种实际意义的数值。

4. 节点的带权路径长度

节点的带权路径长度是指节点的权值和节点的路径长度的乘积。

5. 树的带权路径长度

树的带权路径长度是指树的叶节点的带权路径长度之和。

6. 最优二叉树

最优二叉树是指给定 n 个带有权值的节点作为叶节点构造出的具有最小带权路径长度的二叉树。最优二叉树也叫哈夫曼树。

5.3.2 哈夫曼树的构造

给定 n 个叶节点,它们的权值分别是$\{w_1,w_2,\cdots,w_n\}$,构造相应的哈夫曼树的主要步骤如下。

(1) 构造由 n 棵二叉树组成的森林,每棵二叉树只有一个根节点,根节点的权值分别为$\{w_1,w_2,\cdots,w_n\}$。

(2) 在森林中选取根节点权值最小和次小的两棵二叉树分别作为左子树和右子树去构造一棵新的二叉树,新二叉树的根节点权值为两棵子树的根节点权值之和。

(3) 将两棵二叉树从森林中删除,并将新的二叉树添加到森林中。

(4) 重复步骤(2)和(3),直到森林中只有一棵二叉树,此二叉树即为哈夫曼树。

假设给定的权值为{1,2,3,4,5},图 5.10 展示了哈夫曼树的构造过程。

【例 5.4】 对于给定的一组权值 $W=\{5,2,9,11,8,3,7\}$,试构造相应的哈夫曼树,并计算它的带权路径长度。

解:构造的哈夫曼树如图 5.11 所示。

树的带权路径长度如下:

$$\mathrm{WPL}=2\times4+3\times4+5\times3+7\times3+8\times3+9\times2+11\times2=120$$

5.3.3 哈夫曼编码

在传送信息时需要将信息符号转化成二进制组成的符号串,一般每个字符由一个字节或两个字节表示,即 8 或 16 个位数。为了提高存储和传输效率,需要设计对字符集进行二进制编码的规则,使得利用这种规则对信息进行编码时编码位数最小,即需要传输的信息量最小。

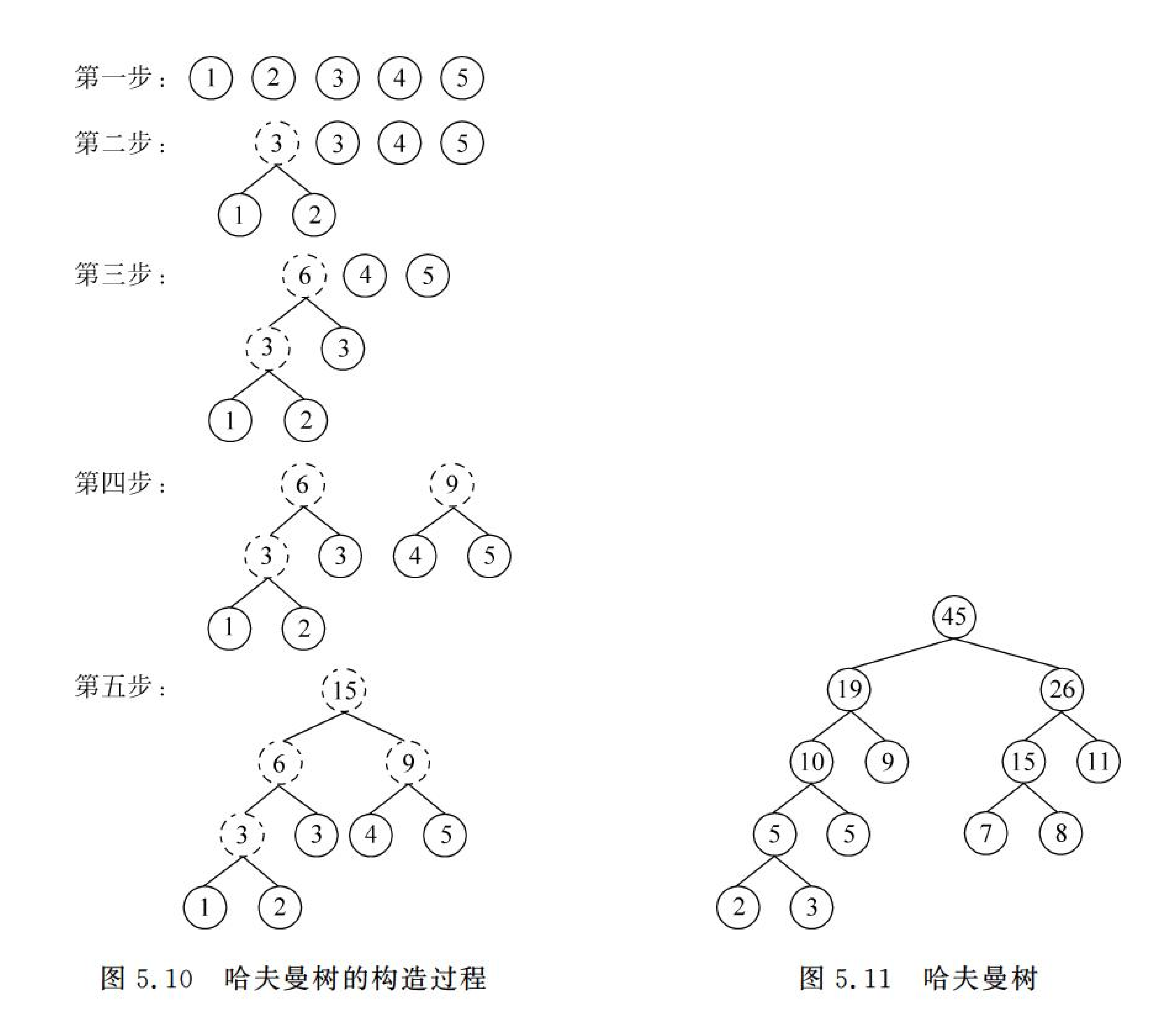


图 5.10 哈夫曼树的构造过程　　　　图 5.11 哈夫曼树

哈夫曼编码是一种不等长的编码方案，数据的编码因其使用频率的不同而长短不一，使用频率高的数据其编码较短，使用频率低的数据其编码较长，从而使所有数据的编码总长度最短。各数据的使用频率通过在全部数据中统计重复数据的出现次数获得。

又因为在编码序列中若使用前缀相同的编码来表示不同的字符会造成二义性，额外的分隔符号会造成传输信息量的增加，为了省去不必要的分隔符号，要求每一个字符的编码都不是另一个字符的前缀，即每个字符的编码都是前缀编码。

利用哈夫曼树构造出的哈夫曼编码是一种最优前缀编码，构造的主要步骤如下。

(1) 对于具有 n 个字符的字符集，将字符的频度作为叶节点的权值，产生 n 个带权叶节点。

(2) 根据 5.3.2 节中介绍的构造哈夫曼树的方法利用 n 个叶节点构造哈夫曼树。

(3) 根据哈夫曼编码规则将哈夫曼树中的每一条左分支标记为 0，每一条右分支标记为 1，则可得到每个叶节点的哈夫曼编码。

哈夫曼编码的译码过程是构造过程的逆过程，从哈夫曼树的根节点开始对编码的每一位进行判别，如果为 0 进入左子树，如果为 1 进入右子树，直到到达叶节点，即译出了一个字符。

5.3.4 构造哈夫曼树和哈夫曼编码的类的描述

构造哈夫曼树需要从子节点到父节点的操作，译码时需要从父节点到子节点的操作，所以为了提高算法的效率将哈夫曼树的节点设计为三叉链式存储结构。一个数据域存储节点



的权值,一个标记域 flag 标记节点是否已经加入到哈夫曼树中,3 个指针域分别存储着指向父节点、孩子节点的地址。

节点类的描述如下:

```
class HuffmanNode(object):
    def __init__(self,data,weight):
        self.data = data           # 节点的值
        self.weight = weight       # 节点的权值
        self.parent = None         # 父节点
        self.lchild = None         # 左孩子
        self.rchild = None         # 右孩子
```

【算法 5.13】 构造哈夫曼树。

```
class HuffmanTree(object):
    def __init__(self,data):
        # data 是编码字符与出现次数的集合,例如 w = [('a',1),('b',2)]
        nodes = [ HuffmanNode(c,w) for c,w in data ]
        self.index = {} # 编码字符的索引
        while len(nodes)>1:
            nodes = sorted(nodes,key = lambda x:x.weight)
            s = HuffmanNode(None,nodes[0].weight + nodes[1].weight)
            s.lchild = nodes[0]
            s.rchild = nodes[1]
            nodes[0].parent = nodes[1].parent = s
            nodes = nodes[2:]
            nodes.append(s)
        self.root = nodes[0]
        self.calIndex(self.root,'') # 递归计算每个字符的哈夫曼编码并保存

    def calIndex(self,root,str):
        if root.data is not None:
            # 保存字符的编码
            self.index[root.data] = str
        else:
            self.calIndex(root.lchild,str + '0')
            self.calIndex(root.rchild,str + '1')

    def queryHuffmanCode(self,c):
        if c not in self.index:
            raise Exception("未编码的字符")
        return self.index[c]
```

【算法 5.14】 若字符与出现频率对应关系如:[('a',5),('b',2),('c',9),('d',11),('e',8),('f',3),('g',7)]求哈夫曼编码。

```
data = [('a',5),('b',2),('c',9),('d',11),('e',8),('f',3),('g',7)]

t = HuffmanTree(data)

```

```
for c,w in data:
    print('字符 %s 的哈夫曼编码为: %s' % (c,t.queryHuffmanCode(c)))
```

输出如下：

```
字符 a 的哈夫曼编码为：010
字符 b 的哈夫曼编码为：0110
字符 c 的哈夫曼编码为：00
字符 d 的哈夫曼编码为：10
字符 e 的哈夫曼编码为：111
字符 f 的哈夫曼编码为：0111
字符 g 的哈夫曼编码为：110
```

【例 5.5】 已知某字符串 s 中共有 8 种字符，各种字符分别出现 2 次、1 次、4 次、5 次、7 次、3 次、4 次和 9 次，对该字符串用[0,1]进行前缀编码，问该字符串的编码至少有多少位？

解：以各字符出现的次数作为叶子节点的权值构造的哈夫曼编码树如图 5.12 所示。其带权路径长度$=2\times5+1\times5+3\times4+5\times3+9\times2+4\times3+4\times3+7\times2=98$，所以该字符串的编码长度至少为 98 位。

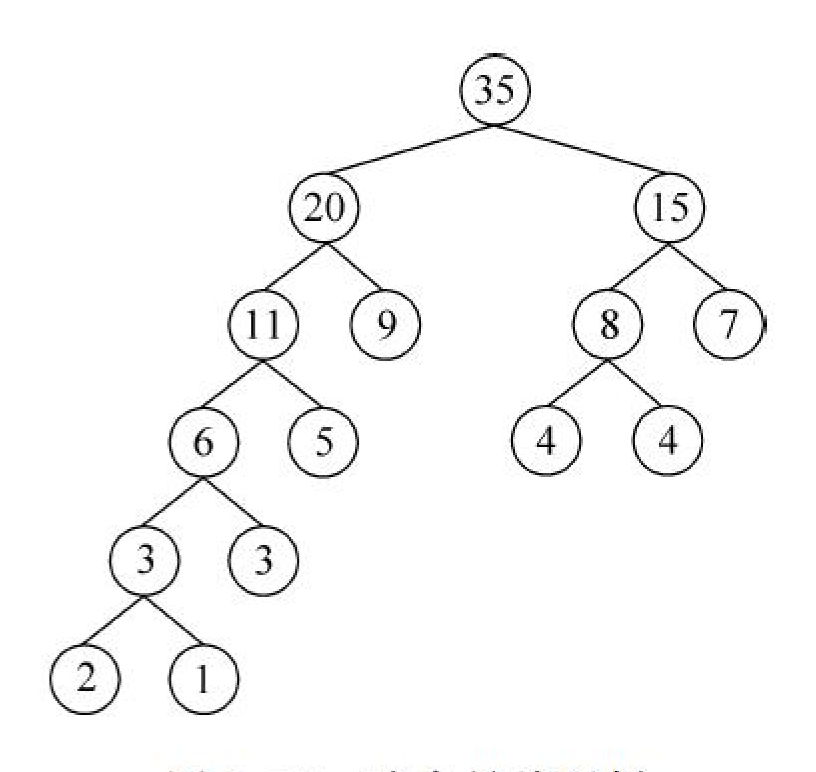


图 5.12 哈夫曼编码树

5.4 树和森林

5.4.1 树的存储结构

一棵树包含各节点间的层次关系和兄弟关系，两种关系的存储结构不同。

树的层次关系必须采用链式存储结构存储，通过链连接父节点和孩子节点。

一个节点的多个孩子节点（互称兄弟节点）之间是线性关系，可以采用顺序存储结构或者链式存储结构。

1. 树的父母孩子链表

树的父母孩子链表采用顺序存储结构存储多个孩子节点，其中 children 数组存储多个孩子节点，各节点的 children 数组元素长度不同，为孩子个数。

2. 树的父母孩子兄弟链表

树的父母孩子兄弟链表采用链式存储结构存储多个孩子节点，节点的 child 链指向一个

孩子节点，sibling 链指向下一个兄弟节点。

森林也可以使用父母孩子兄弟链表进行存储，这种存储结构实际上是把一棵树转换成一棵二叉树存储。其存储规则如下。

(1) 每个节点采用 child 链指向其中一个孩子节点，多个孩子节点之间由 sibling 链连接起来，组成一条具有兄弟节点关系的单链表。

(2) 将每棵树采用树的父母孩子兄弟链表存储。

(3) 森林中的多棵树之间是兄弟关系，将这些树通过根的 sibling 链连接起来。

5.4.2 树的遍历规则

树的孩子优先遍历规则主要有两种，即先序遍历和后序遍历。树的遍历规则也是递归的。

(1) 树的先序遍历：访问根节点；按从左到右的次序遍历根的每一棵子树。

(2) 树的后序遍历：按从左到右的次序遍历根的每一棵子树；访问根节点。

树的层次遍历规则同二叉树。

小　结

(1) 树是数据元素之间具有层次关系的非线性结构，是由 n 个节点构成的有限集合。与线性结构不同，树中的数据元素具有一对多的逻辑关系。

(2) 二叉树是特殊的有序树，它也是由 n 个节点构成的有限集合。当 $n=0$ 时称为空二叉树。二叉树的每个节点最多只有两棵子树，子树也为二叉树，互不相交且有左、右之分，分别称为左二叉树和右二叉树。

(3) 二叉树的存储结构分为两种，即顺序存储结构和链式存储结构。二叉树的顺序存储结构是指将二叉树的各个节点存放在一组地址连续的存储单元中，所有节点按节点序号进行顺序存储；二叉树的链式存储结构是指将二叉树的各个节点随机存放在存储空间中，二叉树的各节点间的逻辑关系由指针确定。

(4) 二叉树具有先序遍历、中序遍历、后序遍历和层次遍历 4 种遍历方式。

(5) 最优二叉树是指给定 n 个带有权值的节点作为叶节点构造出的具有最小带权路径长度的二叉树，也叫哈夫曼树。

(6) 哈夫曼编码是数据压缩技术中的无损压缩技术，是一种不等长的编码方案，使所有数据的编码总长度最短。

习　题　5

一、选择题

1. 如果节点 A 有 3 个兄弟，B 是 A 的双亲，则节点 B 的度是(　　)。

　A. 1　　B. 2　　C. 3　　D. 4

2. 设二叉树有 n 个节点,则其深度为(　　)。

A. $n-1$　　B. n　　C. $\lfloor \log_2 n \rfloor+1$　　D. 不能确定

3. 二叉树的前序序列和后序序列正好相反,则该二叉树一定是(　　)的二叉树。

A. 空或只有一个节点　　B. 高度等于其节点数

C. 任一节点无左孩子　　D. 任一节点无右孩子

4. 线索二叉树中某节点 R 没有左孩子的充要条件是(　　)。

A. R.lchild＝None　　B. R.ltag＝0

C. R.ltag＝1　　D. R.rchild＝None

5. 深度为 k 的完全二叉树最少有(　　)个节点、最多有(　　)个节点,具有 n 个节点的完全二叉树按层序从 1 开始编号,则编号最小的叶子节点的序号是(　　)。

A. $2^{k-2}+1$　　B. 2^k-1　　C. 2^{k-1}　　D. $2^{k-1}-1$

E. 2^{k+1}　　F. $2^{k+1}-1$　　G. $2^{k-1}+1$　　H. 2^k

6. 一个高度为 h 的满二叉树共有 n 个节点,其中有 m 个叶子节点,则(　　)成立。

A. $n=h+m$　　B. $h+m=2n$　　C. $m=h-1$　　D. $n=2m-1$

7. 任何一棵二叉树的叶子节点在前序、中序、后序遍历序列中的相对次序(　　)。

A. 肯定不发生改变　　B. 肯定发生改变

C. 不能确定　　D. 有时发生变化

8. 如果 T' 是由有序树 T 转换而来的二叉树,那么 T 中节点的前序序列就是 T' 中节点的(　　)序列,T 中节点的后序序列就是 T' 中节点的(　　)序列。

A. 前序　　B. 中序　　C. 后序　　D. 层序

9. 设森林中有 4 棵树,树中节点的个数依次为 n_1、n_2、n_3、n_4,则把森林转换成二叉树后其根节点的右子树上有(　　)个节点、根节点的左子树上有(　　)个节点。

A. n_1-1　　B. n_1　　C. $n_1+n_2+n_3$　　D. $n_2+n_3+n_4$

10. 讨论树、森林和二叉树的关系目的是为了(　　)。

A. 借助二叉树上的运算方法去实现对树的一些运算

B. 将树、森林按二叉树的存储方式进行存储并利用二叉树的算法解决树的有关问题

C. 将树、森林转换成二叉树

D. 体现一种技巧,没有什么实际意义

二、填空题

1. 树是 $n(n\geqslant 0)$ 个节点的有限集合,在一棵非空树中有________个根节点,其余节点分成 $m(m>0)$ 个________的集合,每个集合都是根节点的子树。

2. 树中某节点的子树的个数称为该节点的________,子树的根节点称为该节点的________,该节点称为其子树根节点的________。

3. 一棵二叉树的第 $i(i\geqslant 1)$ 层最多有________个节点;一棵有 $n(n>0)$ 个节点的满二叉树共有________个叶子节点和________个非终端节点。

4. 设高度为 h 的二叉树上只有度为 0 和度为 2 的节点,该二叉树的节点数可能达到的最大值是________、最小值是________。

5. 在深度为 k 的二叉树中所含叶子的个数最多为________。

6. 具有 100 个节点的完全二叉树的叶子节点数为________。

7. 已知一棵度为 3 的树有两个度为 1 的节点、3 个度为 2 的节点、4 个度为 3 的节点，则该树中有________个叶子节点。

8. 某二叉树的前序遍历序列是 ABCDEFG，中序遍历序列是 CBDAFGE，则其后序遍历序列是________。

9. 在具有 n 个节点的二叉链表中共有________个指针域，其中________个指针域用于指向其左、右孩子，剩下的________个指针域则是空的。

10. 在有 n 个叶子的哈夫曼树中叶子节点总数为________，分支节点总数为________。

三、算法设计题

1. 设计算法求二叉树的节点个数；按前序次序打印二叉树中的叶子节点；求二叉树的深度。

2. 设计算法判断一棵二叉树是否为完全二叉树。

3. 使用栈将 Tree 类中的递归算法实现为非递归算法。

4. 编写一个非递归算法求出二叉搜索树中的所有节点的最大值，若树为空则返回空值。

5. 编写一个算法求出一棵二叉树中叶子节点的总数，参数初始指向二叉树的根节点。

第6章 图

6.1 图 概 述

在离散数学中，图论研究图的纯数学性质；在数据结构中，图结构研究计算机中如何存储图以及如何实现图的操作和应用。

图是刻画离散结构的一种有力工具。在运筹规划、网络研究和计算机程序流程分析中都存在图的应用问题。我们也经常用图来表达文字难以描述的信息，如城市交通图、铁路网等。

6.1.1 图的基本概念

图是一种数据元素间具有“多对多”关系的非线性数据结构，由顶点集 V 和边集 E 组成，记作 $G=(V,E)$。其中 V 是有穷非空集合，$v\in V$ 称为顶点；E 是有穷集合，$e\in E$ 称为边。

与线性结构和树相比，图更为复杂。从数据间的逻辑关系来说，线性结构的数据元素间存在“一对一”的线性关系；树的数据元素间存在层次关系，具有“一对多”的特性；在图中每一个数据元素都可以和其他的任意数据元素相关。图中的每个元素可以有多个前驱元素和多个后继元素，任意两个元素可以相邻。

下面是有关图的一些基本概念。

1. 无向边

$e=(u,v)$表示顶点 u 和顶点 v 间的一条无向边，也可以简称为边。(u,v)间没有方向，即(u,v)和(v,u)是相同的。

2. 有向边

$e=<u,v>$表示顶点 u 到顶点 v 间的一条有向边，也叫弧。u 叫始点或弧尾，v 叫终点或弧头。$<u,v>$是有方向的，因此$<u,v>$和$<v,u>$是不同的。

3. 零图

零图是指 E 为空集的图，也就是图中只有顶点存在，没有边。

4. 无向图

无向图指全部由无向边构成的图，如图 6.1 所示。

5. 有向图

有向图指全部由有向边构成的图，如图 6.2 所示。

6. 完全图

完全图是指边数达到最大值的图，即在顶点数为 n 的无向图中边数为 $n(n-1)/2$，在顶

点数为 n 的有向图中边数为 $n(n-1)$，如图 6.3 所示。

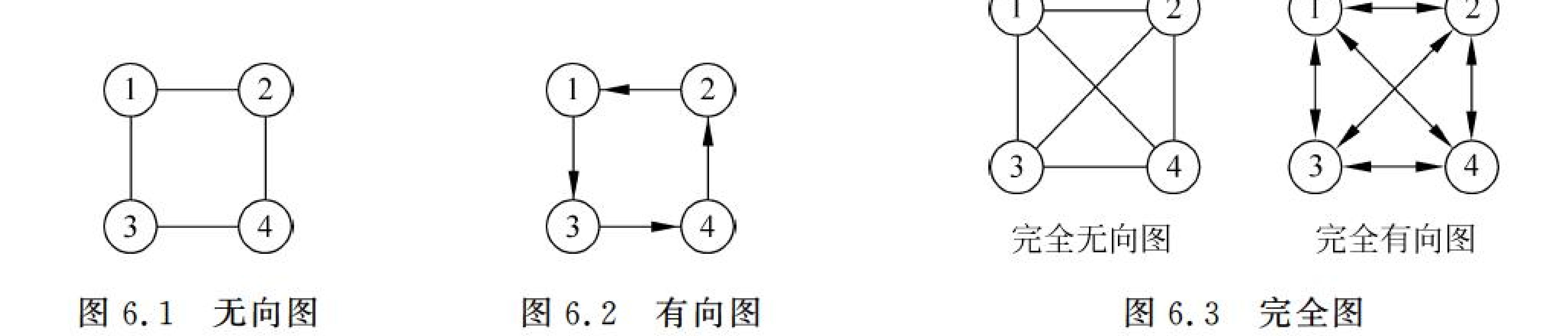


图 6.1　无向图　　图 6.2　有向图　　图 6.3　完全图

7. 稠密图

稠密图是指边数较少的图，如 $e<n\log_2 n$，反之则为稀疏图。

8. 子图

设有两个图 $G=(V,E)$ 和 $G'=(V',E')$，如果有 $V'\subseteq V$ 和 $E'\subseteq E$，则称 G' 是 G 的子图，记作 $G'\subseteq G$。

9. 生成子图

如果 $G'=(V',E')$ 是 $G=(V,E)$ 的子图，并且 $V'=V$，则称 G' 是 G 的生成子图。

10. 邻接点

在一个无向图中若存在边 (u,v)，则称顶点 u 和 v 互为邻接点。边 (u,v) 是顶点 u 和 v 关联的边，顶点 u 和 v 是边 (u,v) 关联的顶点。

在一个有向图中若存在边 $<u,v>$，则称顶点 u 邻接到 v，顶点 v 邻接自 u，弧 $<u,v>$ 与顶点 u 和 v 关联。

11. 顶点的度

顶点的度是指与该顶点关联的边的数目。顶点 u 的度记作 $D(u)$。

在有向图中顶点的度有入度和出度两种。对于顶点 u，入度指的是以 u 为终点的弧的数目，记为 $\mathrm{ID}(u)$；出度指的是以 u 为起点的弧的数目，记为 $\mathrm{OD}(u)$。

全部顶点的度之和为边数的两倍。

12. 路径

路径是指从顶点 u 到顶点 v 所经过的顶点序列。路径长度是指路径上边的数目。没有顶点重复出现的路径叫初等路径。

13. 回路

第一个和最后一个顶点相同的路径称为回路或环，除了第一个和最后一个顶点以外，其他顶点都不重复出现的回路叫初等回路。

14. 连通图

在无向图中若顶点 u 和顶点 v 间有路径，则称 u 和 v 是连通的。连通图是指任意两个顶点均是连通的图。

15. 连通分量

连通分量是指无向图中的极大连通子图。

16. 强连通图

在有向图中若任意两个顶点均是连通的，则称该图为强连通图。

17. 强连通分量

强连通分量是指有向图中的极大连通子图。

18. 生成树和生成森林

生成树是指包含图中的全部顶点，但只有构成树的 $n-1$ 条边的生成子图。对于非连通图，每个连通分量可形成一棵生成树，所有生成树组成的集合叫该非连通图的生成森林。

19. 网

网指的是边上带有权值的图。通常权为非负实数，可以表示从一个顶点到另一个顶点的距离、时间和代价等。

6.1.2 图的抽象数据类型描述

图的抽象数据类型用 Python 抽象类描述如下：

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IGraph(metaclass = ABCMeta):
    @abstractmethod
    def createGraph(self):
        '''创建图'''
        pass
    @abstractmethod
    def getVNum(self):
        '''返回图中的顶点数'''
        pass
    @abstractmethod
    def getENum(self):
        '''返回图中的边数'''
        pass
    @abstractmethod
    def getVex(self,i):
        '''返回位置为 i 的顶点值'''
        pass
    @abstractmethod
    def locateVex(self,x):
        '''返回值为 x 的顶点位置'''
        pass
    @abstractmethod
    def firstAdj(self,i):
        '''返回节点的第一个邻接点'''
        pass
    @abstractmethod
    def nextAdj(self,i,j):
        '''返回相对于 j 的下一个邻接点'''
        pass
```

6.2 图的存储结构

图的存储结构需要存储顶点的值以及与顶点相关联的顶点和边的信息。顶点间没有次序关系，各条边之间也没有次序关系，但是表示和存储一个图必须约定好顶点次序。边集合

表达每对顶点间的邻接关系,是二维线性关系。矩阵的存储结构常见有邻接矩阵、邻接表、十字链表 3 种。

(1) 边采用顺序存储结构,用二维数组存储,称为图的邻接矩阵。

(2) 边采用链式存储结构,存储行的后继,即矩阵行的单链表,称为图的邻接表。

(3) 边采用链式存储结构,存储行和列的后继,即矩阵十字链表,称为图的邻接多重表。

6.2.1 邻接矩阵

1. 图的邻接矩阵的存储结构

假设图 $G=(V,E)$ 具有 n 个顶点,即 $\{v_0, v_1, \cdots, v_{n-1}\}$,那么图的邻接矩阵可定义如下:

$$\mathbf{A}[i][j]=\begin{cases}1, & < v_i, v_j > \in \ \#E \text{ 或} (v_i, v_j) \in E \\ 0, & < v_i, v_j > \notin \ \#E \text{ 且} (v_i, v_j) \notin E\end{cases}$$

其中 $0 \leqslant i, j < n$。

假设图 $G=(V,E)$ 为网,且 w_{ij} 为边 (v_i, v_j) 或 $< v_i, v_j >$ 上的权值,则网的邻接矩阵可定义如下:

$$\mathbf{A}[i][j]=\begin{cases}w_{ij}, & < v_i, v_j > \in \ \#E \text{ 或} (v_i, v_j) \in E \\ \infty, & < v_i, v_j > \notin \ \#E \text{ 且} (v_i, v_j) \notin E\end{cases}$$

其中 $0 \leqslant i, j < n$。

分析可得,在无向图的邻接矩阵中第 i 行或者第 i 列的非零元素的个数为第 i 个顶点的度;在有向图的邻接矩阵中第 i 行的非 ∞ 元素的个数为第 i 个顶点的出度,第 i 列非 ∞ 元素的个数为第 i 个顶点的入度。无向图的邻接矩阵是对称的,有向图的邻接矩阵不一定是对称的。

图的邻接矩阵可以用二维数组进行表示,邻接矩阵类的 Python 语言描述如下:

```
class MGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind              # 图的种类
        self.vNum = vNum              # 图的顶点数
        self.eNum = eNum              # 图的边数
        self.v = v                    # 顶点列表
        self.e = e                    # 邻接矩阵
```

2. 图的邻接矩阵类的基本操作的实现

1) 图的创建

【算法 6.1】 创建无向图。

```
def createUDG(self,vNum,eNum,v,e):
    self.vNum = vNum
```

```
    self.eNum = eNum
    self.v = [None] * vNum                # 构造顶点集
    for i in range(vNum):
        self.v[i] = v[i]
    self.e = [ [0] * vNum ] * vNum        # 构造边集
    for i in range(eNum):
        a,b = e[i]
    m,n = self.locateVex(a),self.locateVex(b)
    self.e[m][n] = self.e[n][m] = 1
```

【算法 6.2】 创建有向图。

```
def createDG(self,vNum,eNum,v,e):
    self.vNum = vNum
    self.eNum = eNum
    self.v = [None] * vNum                # 构造顶点集
    for i in range(vNum):
        self.v[i] = v[i]
    self.e = [ [0] * vNum ] * vNum        # 构造边集
    for i in range(eNum):
        a,b = e[i]
    m,n = self.locateVex(a),self.locateVex(b)
    self.e[m][n] = 1
```

【算法 6.3】 创建无向网。

```
def createUDN(self,vNum,eNum,v,e):
    self.vNum = vNum
    self.eNum = eNum
    self.v = [None] * vNum                    # 构造顶点集
    for i in range(vNum):
        self.v[i] = v[i]
    self.e = [ [sys.maxsize] * vNum ] * vNum  # 初始化边集
    for i in range(eNum):
        a,b,w = e[i]
    m,n = self.locateVex(a),self.locateVex(b)
    self.e[m][n] = self.e[n][m] = w
```

【算法 6.4】 创建有向网。

```
def createDN(self,vNum,eNum,v,e):
    self.vNum = vNum
    self.eNum = eNum
    self.v = [None] * vNum                          # 构造顶点集
    for i in range(vNum):
        self.v[i] = v[i]
    self.e = [ [sys.maxsize] * vNum ] * vNum        # 初始化边集
    for i in range(eNum):
        a,b,w = e[i]
    m,n = self.locateVex(a),self.locateVex(b)
    self.e[m][n] = w
```

2）顶点的定位

顶点定位算法 locateVex(x)是根据 x 的值取得其在顶点集中的位置，若不存在则返回 −1。

【算法 6.5】 顶点的定位。

```
def locateVex(self,x):
    for i in range(self.vNum):
        if self.v[i] == x:
            return i
    return -1
```

3）查找第一个邻接点

查找第一个邻接点算法 firstAdj(i)是指给定一个顶点在顶点集中的位置 i，返回其第一个邻接点，若不存在则返回 −1。

【算法 6.6】 查找第一个邻接点。

```
def firstAdj(self,i):
    if i<0 or i>=self.vNum:
        raise Exception("第%s个顶点不存在" % i)
    for j in range(self.vNum):
        if self.e[i][j]!= 0 and self.e[i][j]< sys.maxsize:
            return j
    return -1
```

4）查找下一个邻接点

查找下一个邻接点算法 nextAdj(i,j)是指给定两个顶点在顶点集中的位置 i、j，第 j 个顶点是第 i 个顶点的邻接点，返回第 j 个顶点之后的下一个邻接点，若不存在则返回 −1。

【算法 6.7】 查找下一个邻接点。

```
def nextAdj(self,i,j):
    if j == self.vNum-1:
        return -1
    for k in range(j+1,self.vNum):
        if self.e[i][k]!= 0 and self.e[i][k]< sys.maxsize:
            return k
    return -1
```

3. 邻接矩阵表示图的性能分析

图的邻接矩阵表示存储了任意两个顶点间的邻接关系或边的权值，能够实现对图的各种操作，其中判断两个顶点间是否有边相连、获得和设置边的权值等操作的时间复杂度为 $O(1)$。但是，与顺序表存储线性表的性能相似，由于采用数组存储，每插入或者删除一个元素需要移动大量元素，使得插入和删除操作的效率很低，而且数组容量有限，当扩充容量时需要复制全部元素，效率更低。

在图的邻接矩阵中每个矩阵元素表示两个顶点间的邻接关系，无边或有边。即使两个顶点之间没有邻接关系，也占用一个存储单元存储 0 或者 −1。对于一个有 n 个顶点的完全图，其邻接矩阵有 $n(n-1)/2$ 个元素，此时邻接矩阵的存储效率较高；当图中的边数较少时，邻接矩阵变得稀疏，存储效率较低，此时可用图的邻接表进行存储。

【**例 6.1**】 n 个顶点的无向图采用邻接矩阵存储，回答下列问题：

(1) 图中有多少条边？

(2) 任意两个顶点 i 和 j 之间是否有边相连？

(3) 任意一个顶点的度是多少？

解：

(1) 邻接矩阵中非零元素个数的总和除以 2。

(2) 当邻接矩阵 A 中 $A[i][j]=1$(或 $A[j][i]=1$)时表示两顶点之间有边相连。

(3) 计算邻接矩阵上该顶点对应的行上非零元素的个数。

6.2.2 邻接表

1. 图的邻接表存储结构

邻接表采用链式存储结构存储图，是由一个顺序存储的顶点表和多个链式存储的边表组成的。边表的个数和图的顶点数相同。顶点表由顶点节点组成，每个顶点节点又由数据域和指针域组成，其中数据域 data 存放顶点值，指针域 firstArc 指向边表中的第一个边节点。边表由边节点组成，每个边节点又由 adjVex、nextArc、value 几个域组成，其中 value 存放边的信息，例如权值；adjVex 存放与节点邻接的顶点在图中的位置；nextArc 指向下一个边节点。

邻接表的顶点节点类的 Python 描述如下：

```
class VNode(object):
    def __init__(self,data = None,firstNode = None):
        self.data = data                  # 存放节点值
        self.firstArc = firstNode         # 第一条边
```

邻接表的边节点类的 Python 描述如下：

```
class ArcNode(object):
    def __init__(self,adjVex,value,nextArc = None):
        self.adjVex = adjVex          # 边指向的顶点的位置
        self.value = value            # 边的权值
        self.nextArc = nextArc        # 指向下一条边
```

图的邻接表类的描述如下：

```
class ALGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind # 图的种类
        self.vNum = vNum # 图的顶点数
        self.eNum = eNum # 图的边数
        self.v = v # 顶点列表
```

```
            self.e = e # 边信息

    def createGraph(self):
        if self.kind == self.GRAPHKIND_UDG:
            self.createUDG()
        elif self.kind == self.GRAPHKIND_DG:
            self.createDG()
        elif self.kind == self.GRAPHKIND_UDN:
            self.createUDN()
        elif self.kind == self.GRAPHKIND_DN:
            self.createDN()

    def createUDG(self):
        '''创建无向图'''
        pass

    def createDG(self):
        '''创建有向图'''
        pass

    def createUDN(self):
        '''创建无向网'''
        pass

    def createDN(self):
        '''创建有向网'''
        pass

    def addArc(self, i,j,value):
        '''插入边节点'''
        pass

    def firstAdj(self,i):
        '''查找第一个邻接点'''
        pass

    def nextAdj(self,i,j):
        '''返回 i 相对于 j 的下一个邻接点'''
        pass

    def getVNum(self):
        '''返回顶点数'''
        return self.vNum

    def getENum(self):
        '''返回边数'''
        return self.eNum

    def getVex(self,i):
        '''返回第 i 个顶点的值'''
```

```
        if i < 0 or i >= self.vNum:
            raise Exception("第%s个顶点不存在" % i)
        return self.v[i].data

    def locateVex(self,x):
        '''返回值为x的顶点的位置'''
        for i in range(self.vNum):
            if self.v[i].data == x:
                return i
        return -1

    def getArcs(self,u,v):
        '''返回顶点u到顶点v的距离'''
        if u < 0 or u >= self.vNum:
            raise Exception("第%s个节点不存在" % u)
        if v < 0 or v >= self.vNum:
            raise Exception("第%s个节点不存在" % v)
        p = self.v[u].firstArc
        while p is not None:
            if p.adjVex == v:
                return p.value
            p = p.nextArc
        return sys.maxsize
```

2. 图的邻接表的基本操作的实现

1）图的创建

【算法 6.8】 创建无向图。

```
def createUDG(self):
    '''创建无向图'''
    v = self.v
    self.v = [ None ] * self.vNum
    for i in range(self.vNum):
        self.v[i] = VNode(v[i])
    for i in range(self.eNum):
        a,b = self.e[i]
        u,v = self.locateVex(a),self.locateVex(b)
        self.addArc(u,v,1)
        self.addArc(v,u,1)
```

【算法 6.9】 创建有向图。

```
def createDG(self):
    '''创建有向图'''
    v = self.v
    self.v = [ None ] * self.vNum
    for i in range(self.vNum):
        self.v[i] = VNode(v[i])
    for i in range(self.eNum):
        a,b = self.e[i]
```

```
        u,v = self.locateVex(a),self.locateVex(b)
        self.addArc(u,v,1)
```

【算法 6.10】 创建无向网。

```
def createUDN(self):
    '''创建无向网'''
    v = self.v
    self.v = [ None ] * self.vNum
    for i in range(self.vNum):
        self.v[i] = VNode(v[i])
    for i in range(self.eNum):
        a,b,w = self.e[i]
        u,v = self.locateVex(a),self.locateVex(b)
        self.addArc(u,v,w)
        self.addArc(v,u,w)
```

【算法 6.11】 创建有向网。

```
def createDN(self):
    '''创建有向网'''
    v = self.v
    self.v = [ None ] * self.vNum
    for i in range(self.vNum):
        self.v[i] = VNode(v[i])
    for i in range(self.eNum):
        a,b,w = self.e[i]
        u,v = self.locateVex(a),self.locateVex(b)
        self.addArc(u,v,w)
```

2) 在图中插入边节点

插入边节点的算法 addArc(i,j,value)是指在边链表中加入一个由第 i 个顶点指向第 j 个顶点的权值为 value 的边节点,采用头插法进行插入。

【算法 6.12】 插入边节点。

```
def addArc(self, i,j,value):
    '''插入边节点'''
    arc = ArcNode(j,value)
    arc.nextArc = self.v[i].firstArc
    self.v[i].firstArc = arc
```

3) 查找第一个邻接点

【算法 6.13】 查找第一个邻接点。

```
def firstAdj(self,i):
    '''查找第一个邻接点'''
    if i<0 or i>=self.vNum:
        raise Exception("第%s个节点不存在" % i)
    p = self.v[i].firstArc
    if p is not None:
        return p.adjVex
```

```
    return -1
```

4）查找下一个邻接点

```
def nextAdj(self,i,j):
    '''返回 i 相对于 j 的下一个邻接点'''
    if i<0 or i>= self.vNum:
        raise Exception("第%s个节点不存在" % i)
    p = self.v[i].firstArc
    while p is not None:
        if p.adjVex == j:
            break
        p = p.nextArc
    if p.nextArc is not None:
        return p.nextArc.adjVex
    return -1
```

用邻接矩阵存储图可以很好地确定两个顶点间是否有边，但是查找顶点的邻接点，需要访问对应一行或一列的所有数据元素，并且无论两个顶点间是否有边都要保留存储空间。

用邻接表存储图可以方便地找到顶点的邻接点，对于稀疏图来说节省存储空间，但若要确定两个顶点间是否有边相连则需要遍历单链表，比邻接矩阵复杂。

6.3 图 的 遍 历

图的遍历是指从图的任意一个顶点出发对图的每个顶点访问且仅访问一次的过程，因为图中可能存在回路，为了避免对一个顶点的重复访问可以增设一个辅助数组 visited[0..$n-1$]，全部初始化为 0，一旦第 i 个顶点被访问，置 visited[i]=1。图的遍历和树的遍历相比更加复杂，需要考虑以下 3 个问题。

(1) 指定遍历的第一个顶点。

(2) 由于一个顶点和多个顶点相邻，需要在多个邻接顶点间确定访问次序。

(3) 由于图中存在回路，必须对访问过的顶点做标记，防止出现重复访问同一顶点的情况。

图的遍历方式分为深度优先搜索和广度优先搜索两种。

广度优先搜索是一种分层的搜索过程，以一个顶点 u 为起始点访问其邻接点 v_0，v_1，…，然后按顺序访问 v_0，v_1，…的各邻接点，重复此过程，即依次访问和顶点 u 之间存在路径并且路径长度为 1，2，…的顶点。

1. 图的广度优先搜索算法

图的广度优先搜索遵循"先被访问的顶点，其邻接点先被访问"规则，因此可引入队列。先将起始点加入队列中，以后每次从队列中删除一个数据元素，依次访问它的未被访问的邻接点，并将其插入到队列中，直到队列为空。

其主要步骤如下。

(1) 建立访问标识数组 visited[n]并初始化为 0，n 为图顶点的个数。

(2) 将未访问顶点 v_i 入队。

(3) 将队首元素顶点 v_i 从队列中取出,依次访问它的未被访问的邻接点 v_j,v_k,…,并将其入队。

(4) 重复步骤(3),直到队列为空。

(5) 改变 i 值,$0\leqslant i<n$,跳到步骤(2)继续进行,直到 $i=n-1$。

【算法 6.14】 图的广度优先搜索。

```
def BFSTraverse(g):
    global visited
    # 建立访问标志数组
    visited = [ False ] * g.getVNum()
    # 以每个顶点作为起始顶点进行遍历
    for i in range(g.getVNum()):
        if not visited[i]:
            BFS(g,i)

def BFS(g,i):
    # 建立辅助队列
    q = LinkQueue()
    # 将起点加入队列
    q.offer(i)
    while not q.isEmpty():
        u = q.poll()
        # 标记顶点已访问
        visited[u] = True
        print(g.getVex(u),end = ' ')
        v = g.firstAdj(u)
        while v =- 1:
            # 顶点未访问的邻接点入队
            if not visited[v]:
                q.offer(v)
            # 获取下一个邻接点
            v = g.nextAdj(u,v)
```

假设图有 n 个顶点和 m 条边,当图的存储结构是邻接矩阵时需要扫描邻接矩阵的每一个顶点,其时间复杂度为 $O(n^2)$;当图的存储结构是邻接表时需要扫描每一条单链表,其时间复杂度为 $O(e)$。

2. 图的深度优先搜索算法

深度优先搜索类似于树的先序遍历,以一个顶点 u 为起始点访问其邻接点 v_0,再访问 v_0 的未被访问的邻接点 v_1,然后从 v_1 出发继续进行类似的访问,直到所有的邻接点都被访问。然后后退到前一个被访问的顶点,看是否有其他未被访问的顶点,若有再进行类似的访问,若无继续回退,直到图中的所有顶点都被访问为止。

其主要步骤如下。

(1) 建立访问标识数组 visited[n]并初始化为 0,n 为图顶点的个数。

(2) 以未访问顶点 v_i 为起始点访问其未访问邻接点 v_j。

(3) 从 v_j 出发递归进行步骤(2),直到所有邻接点均被访问。

(4) 改变 i 值,$0\leqslant i<n$,跳到步骤(2)继续进行,直到 $i=n-1$。

【**算法 6.15**】 图的深度优先搜索。

```
def DFSTraverse(g):
    global visited
    # 建立访问标志数组
    visited = [ False ] * g.getVNum()
    # 以每个顶点作为起始顶点进行遍历
    for i in range(g.getVNum()):
        if not visited[i]:
            DFS(g,i)

def DFS(g,i):
    visited[i] = True
    print(g.getVex(i),end = '')
    v = g.firstAdj(i)
    while v!=-1:
        if not visited[v]:
            DFS(g,v)
        v = g.nextAdj(i,v)
```

假设图有 n 个顶点和 m 条边,当图的存储结构是邻接矩阵时需要扫描邻接矩阵的每一个顶点,其时间复杂度为 $O(n^2)$;当图的存储结构是邻接表时需要扫描每一条单链表,其时间复杂度为 $O(e)$。

【**例 6.2**】 编程利用广度优先搜索算法确定无向图的连通分量。

解:

```
def BFSTraverse(g):
    global visited
    # 建立访问标志数组
    visited = [ False ] * g.getVNum()
    count = 0
    # 以每个顶点作为起始顶点进行遍历
    for i in range(g.getVNum()):
        if not visited[i]:
            count += 1
            print("第%s个连通块:" % count,end = '')
            BFS(g,i)
            print()

def BFS(g,i):
    # 建立辅助队列
    q = LinkQueue()
    # 将起点加入队列
    q.offer(i)
    while not q.isEmpty():
        u = q.poll()
        # 标记顶点已访问
        visited[u] = True
        print(g.getVex(u),end = '')
        v = g.firstAdj(u)
```

```
            while v!=-1:
                # 顶点未访问的邻接点入队
                if not visited[v]:
                    q.offer(v)
                # 获取下一个邻接点
                v = g.nextAdj(u,v)

v= [1,2,3,4,5,6]
e= [(1,2),(2,3),(4,5)]
g= ALGraph(ALGraph.GRAPHKIND_UDG, len(v), len(e), v, e)
g.createGraph()
BFSTraverse(g)
```

输出如下：

```
第 1 个连通块: 1 2 3
第 2 个连通块: 4 5
第 3 个连通块: 6
```

【例 6.3】 已知一个连通图如图 6.4 所示,试给出图的邻接矩阵和邻接表存储示意图,若从顶点 v_1 出发对该图进行遍历,分别给出一个按深度优先遍历和广度优先遍历的顶点序列。

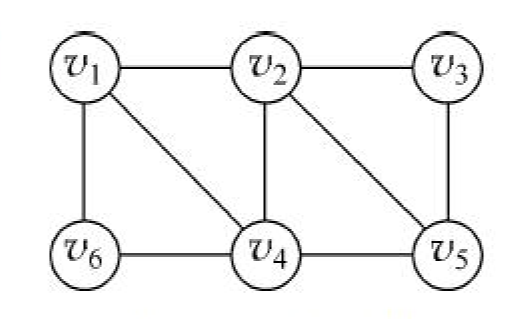
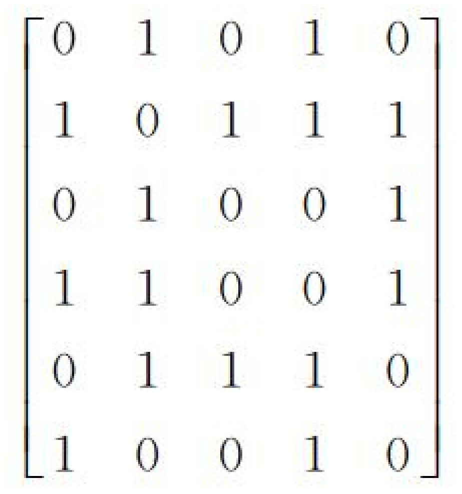


图 6.4　连通图

解：



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

深度优先遍历序列：v_1 v_2 v_3 v_4 v_5 v_6

广度优先遍历序列：v_1 v_2 v_4 v_6 v_3 v_5

邻接表表示如图 6.5 所示。

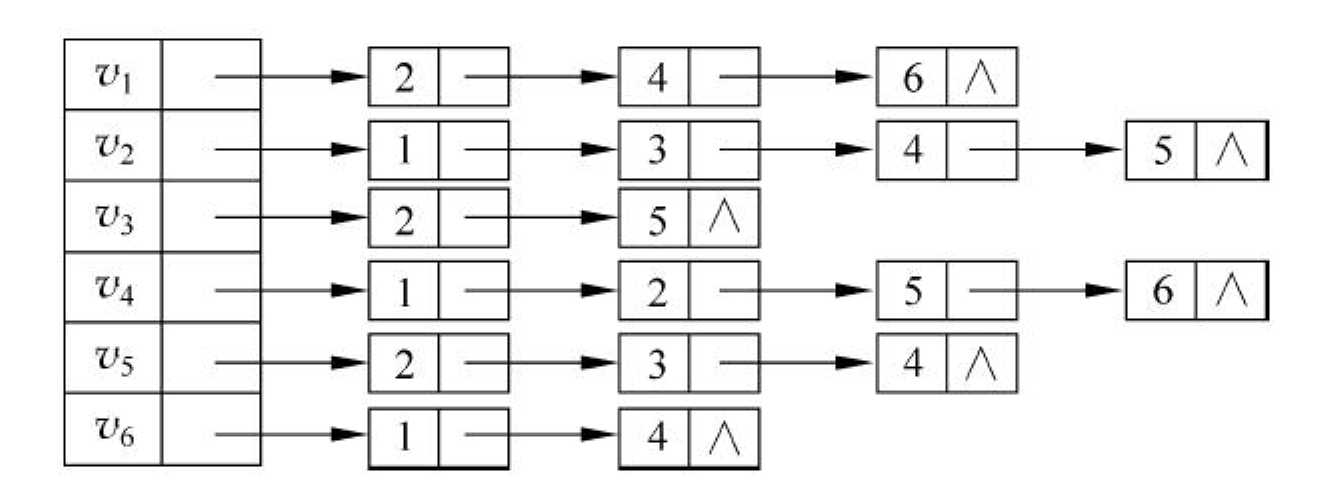


图 6.5　邻接表

【例 6.4】 已知无向图 G 的邻接表如图 6.6 所示,分别写出从顶点 1 出发的深度遍历和广度遍历序列,并画出相应的生成树。

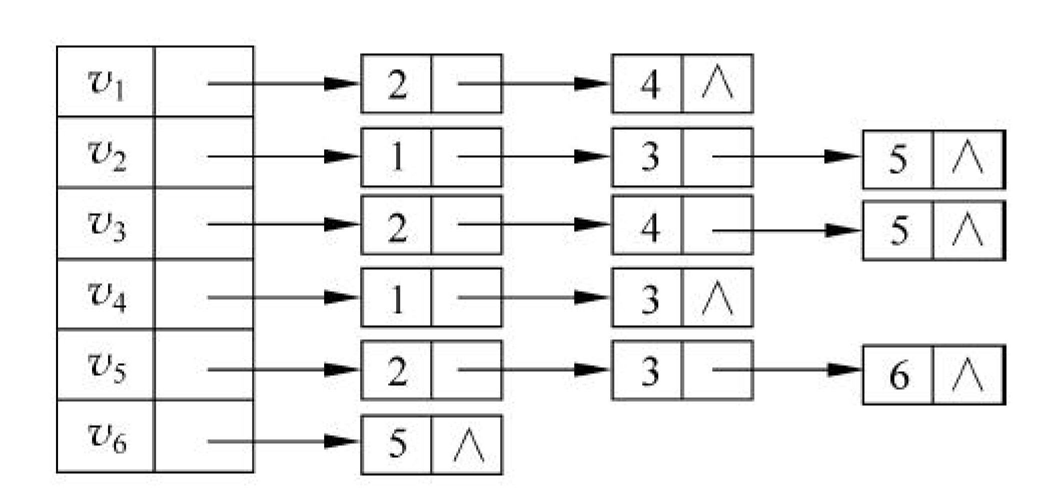


图 6.6　无向图 G 的邻接表

解：

由本章给出的算法可知深度优先遍历序列为 1,2,3,4,5,6。

对应的生成树如图 6.7 所示。

广度优先遍历序列为 1,2,4,3,5,6。

对应的生成树如图 6.8 所示。

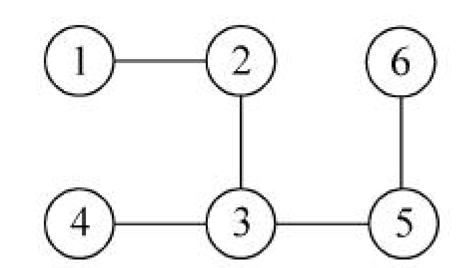


图 6.7　深度优先遍历序列对应的生成树

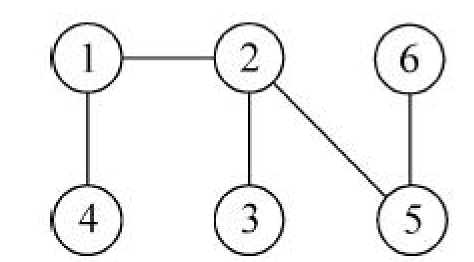


图 6.8　广度优先遍历序列对应的生成树

【例 6.5】　假设有如图 6.9 所示的有向网图，利用 Dijkstra 算法求从顶点 v_1 到其他各顶点的最短路径。

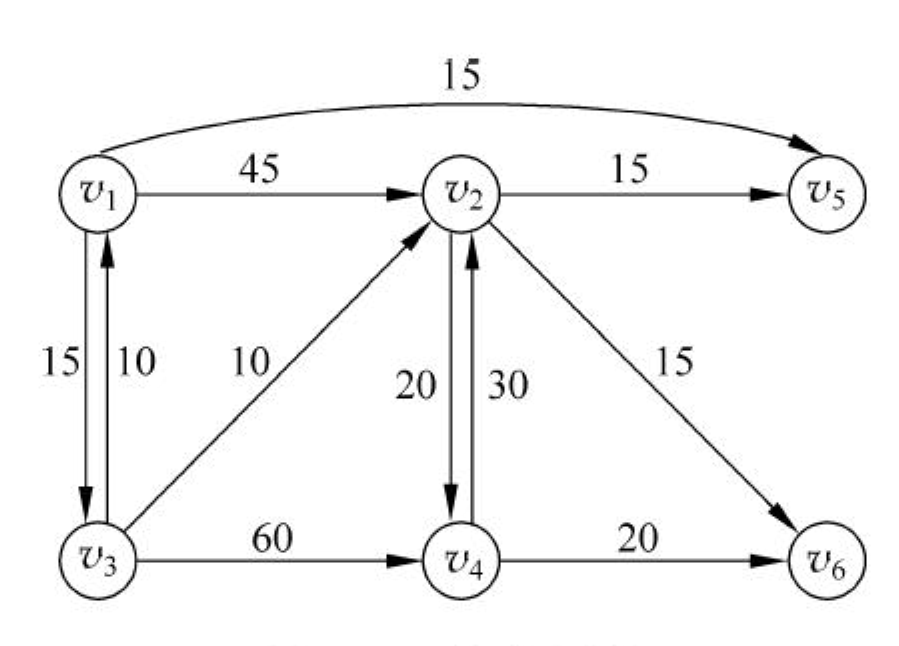


图 6.9　有向网图

解：

从源点 v_1 到其他各顶点的最短路径如表 6.1 所示。

表 6.1　源点

源点	终点	最短路径	最短路径长度
v_1	v_3	$v_1 v_3$	15
v_1	v_5	$v_1 v_5$	15
v_1	v_2	$v_1 v_3 v_2$	25
v_1	v_6	$v_1 v_3 v_2 v_6$	40
v_1	v_4	$v_1 v_3 v_2 v_4$	45

6.4 最小生成树

6.4.1 最小生成树的基本概念

连通图的生成树是图的极小连通子图,它包含图中的全部顶点,但只有构成一棵树的边。一个有 n 个顶点的连通图的生成树只有 $n-1$ 条边。若有 n 个顶点而少于 $n-1$ 条边,则为非连通图,若多于 $n-1$ 条边,则一定形成回路。

由广度优先遍历和深度优先遍历得到的生成树分别称为广度优先生成树和深度优先生成树。根据遍历方法的不同或遍历起点的不同得到的生成树也是不同的,图的生成树不唯一。

对于非连通图,每个连通分量中的顶点集和遍历经过的边一起构成若干棵生成树,共同组成了该非连通图的生成森林。

在一个网的所有生成树中权值总和最小的生成树称为最小代价生成树,简称为最小生成树,最小生成树不一定唯一,需要满足以下 3 条准则。

(1) 只能使用图中的边构造最小生成树。

(2) 具有 n 个顶点和 $n-1$ 条边。

(3) 不能使用产生回路的边。

产生最小生成树的方法主要有 Kruskal 算法和 Prim 算法两种。

6.4.2 Kruskal 算法

Kruskal 算法是依次找出权值最小的边建立最小生成树,每次新增的边不能使生成树产生回路,直到找到 $n-1$ 条边。

设图 T 是由 n 个顶点组成的连通无向网,是图 G 的最小生成树,其中 V 是 T 的顶点集,TE 是 T 的边集。构造最小生成树的步骤如下。

(1) 将 T 的初始状态置为仅含有源点的集合。

(2) 在图 G 的边集中选取权值最小的边,若该边未使生成树 T 形成回路,则加入到 TE 中,否则丢弃,直到生成树中包含了 $n-1$ 条边。

Kruskal 算法的执行时间主要取决于图的边数,时间复杂度为 $O(n^2)$,因此该算法适用于稀疏图的操作。

6.4.3 Prim 算法

在介绍 Prim 算法之前需要先了解距离的概念。

(1) 两个顶点之间的距离:将顶点 u 邻接到顶点 v 的关联边的权值,记为 $|u,v|$。若两个顶点之间不相连,则这两个顶点之间的距离为无穷大。

(2) 顶点到顶点集合的距离:顶点到顶点集合中所有顶点的距离的最小值,记为 $|u,V|=\min|u,v|$。

(3) 两个顶点集合之间的距离:顶点集合 U 的顶点到顶点集合 V 的距离的最小值,记为 $|U,V|=\min|u,V|$。

设图 T 是由 n 个顶点组成的连通无向网，是图 G 的最小生成树，其中 V 是 T 的顶点集，TE 是 T 的边集，构造最小生成树的步骤为从源点开始，必存在一条边，使得 U、V 之间的距离最小，将加入到集合 TE 中，同时将顶点加入到顶点集 U 中，直到 $U=V$ 为止。

针对每一个顶点引入分量 closedge[i]，它包含两个域，lowcost 域存储该边上的权值，即顶点到顶点集 U 的距离；adjvex 域存储该边在顶点集 U 中的顶点。因为集合 U 是随着数据元素的加入而逐渐增大的，所以有新的数据元素加入时将 closedge[i].lowcost 与到新的数据元素的距离进行比较即可。

Prim 算法构造最小生成树的类描述如下：

```
class CloseEdge(object):
    def __init__(self,adjVex,lowCost):
        self.adjVex = adjVex # 在结合 U 中的顶点的值
        self.lowCost = lowCost  # 到集合 U 的最小距离

class MiniSpanTree(object):

    def PRIM(g,u):
        '''从值为 u 的顶点出发构造最小生成树,返回由生成树边组成的二维数组'''
        # 存放生成树边上的顶点
        tree = [ [None,None] for i in range(g.getVNum() - 1) ]
        count = 0
        closeEdge = [ None ] * g.getVNum()
        k = g.locateVex(u)
        for j in range(g.getVNum()):
            if k!= j:
                closeEdge[j] = CloseEdge(u,g.getArcs(k,j))
        # 将 u 添加到 U 中
        closeEdge[k] = CloseEdge(u,0)

        for i in range(1,g.getVNum()):
            # 找出具有到集合 U 最小距离的顶点的序号
            k = MiniSpanTree.getMinMum(closeEdge)
            tree[count][0] = closeEdge[k].adjVex # 在集合 U 中的顶点
            tree[count][1] = g.getVex(k) # 在集合 V - U 中的顶点
            count += 1
            # 更新 closeEdge
            closeEdge[k].lowCost = 0
            for j in range(g.getVNum()):
                if g.getArcs(k,j)< closeEdge[j].lowCost:
                    closeEdge[j] = CloseEdge(g.getVex(k),g.getArcs(k,j))
        return tree

    def getMinMum(closeEdge):
        '''选出 lowcost 最小的顶点'''
        minvalue = sys.maxsize
        v = - 1
        for i in range(len(closeEdge)):
            if closeEdge[i].lowCost!= 0 and closeEdge[i].lowCost < minvalue:
```

```
                minvalue = closeEdge[i].lowCost
                v = i
        return v
```

【例 6.6】 编写程序实现如图 6.10 所示的连通无向网的最小生成树。

图 6.10 连通无向网

解：

```
v= ['A','B','C','D','E','F']
e= [
    ('A','B',7),('A','C',5),('A','D',1),
    ('B','D',6),('B','E',3),
    ('C','D',7),('C','F',2),
    ('D','E',6),('D','F',4),
    ('E','F',7),
]
g= ALGraph(ALGraph.GRAPHKIND_UDN,len(v),len(e),v,e)
g.createGraph()
print(MiniSpanTree.PRIM(g,'A'))
```

输出：

```
[['A', 'D'], ['D', 'F'], ['F', 'C'], ['D', 'B'], ['B', 'E']]
```

6.5 最短路径

最短路径的求解问题主要分为两类，即求某个顶点到其余顶点的最短路径，以及求每一对顶点间的最短路径，本节针对这两类问题提出了两种算法。

6.5.1 单源最短路径

针对这一最短路径问题本节提出了 Dijkstra 算法，基本思想是“按最短路径长度递增的次序”产生最短路径。

若从源点到某个终点存在路径，则一定存在最短路径。从源点到其余各顶点的最短路径长度不一定一样，具有以下特点。

(1) 在这些最短路径中长度最短的最短路径一定有且仅有一条弧，弧的权值是从源点出发的所有弧的权中的最小值。

(2) 长度次短的最短路径有两种情况：其一。只包含一条从源点出发的弧，弧上的权值大于已求得最短路径的弧的权值，小于其他从源点出发的弧的权值；其二，一条只经过已求得最短路径的顶点的路径。

算法的主要步骤为保存当前已经得到的从源点到各个其余顶点的最短路径，也就是说，若源点到该顶点有弧，存在一条路径，长度为弧上的权值，每求得一条到达某个顶点的最短路径就需要检查是否存在经过这个顶点的其他路径，若存在，判断其长度是否比当前求得的路径长度短，若是，则修改当前路径。在算法中引入一个辅助向量 $\boldsymbol{D}$，它的每个分量 $\boldsymbol{D}[i]$ 存

放当前所找到的从源点到终点的最短路径长度。

Dijkstra 算法构造最短路径的类 Python 语言描述如下：

```
class ShortestPath(object):
    def Djikstra(g,v0):
        # 存放最短路径,p[v][k]表示从 v0 到 v 的最短路径中经过的第 k 个点
        p = [ ([ -1 ] * g.getVNum()) for i in range(g.getVNum()) ]
        # 存放最短路径长度
        D = [ sys.maxsize ] * g.getVNum()
        # 若已找到最短路径,finish[v]为 True
        finish = [ False ] * g.getVNum()
        v0 = g.locateVex(v0)
        for v in range(g.getVNum()):
            D[v] = g.getArcs(v0,v)
            if D[v]< sys.maxsize:
                # 从起点直接可以到达
                p[v][0] = v0
                p[v][1] = v
        p[v0][0] = v0 # 起点本身可以直接到达
        D[v0] = 0
        finish[v0] = True
        v = -1
        for i in range(1,g.getVNum()):
            minvalue = sys.maxsize
            for w in range(g.getVNum()):
                # 找出所有最短路径中的最小值
                if not finish[w]:
                    if D[w]< minvalue:
                        v = w
                        minvalue = D[w]
            finish[v] = True
            # 更新当前的最短路径
            for w in range(g.getVNum()):
                if not finish[w] and g.getArcs(v,w)< sys.maxsize and (minvalue + g.getArcs(v,w)< D[w]):
                    D[w] = minvalue + g.getArcs(v,w)
                    for k in range(g.getVNum()):
                        p[w][k] = p[v][k]
                        if p[w][k] == -1:
                            p[w][k] = w
                            break
        dis = { g.getVex(i):D[i] for i in range(g.getVNum()) }
        # 返回到各点最短路的字典与路径矩阵
        return dis,p

    def printDjikstraPath(g,v0,p):
        # v0 到各点的输出最短路径,即 p[v][0]到 p[v][j]直到 p[v][j] ==-1
        u = v0
        v0 = g.locateVex(v0)
        for i in range(g.getVNum()):
```

```
        v = g.getVex(i)
        print('%s->%s的最短路径为:' % (u,v),end='')
        if p[i][0]!=-1:
            print(g.getVex(p[v0][0]),end='')
        for k in range(1,g.getVNum()):
            if p[i][k]==-1:
                break
            print('->%s' % g.getVex(p[i][k]),end='')
        print()
```

分析可得,Dijkstra 算法的时间复杂度为 $O(n^2)$,并且找到一条从源点到某一特定终点之间的最短路径,和求从源点到各个终点的最短路径一样复杂,时间复杂度也为 $O(n^2)$。

6.5.2 求任意两个顶点间的最短路径

求任意两个顶点间的最短路径,如果使用 Dijkstra 算法,可以依次将顶点设为源点,调用算法 n 次即可求得,时间复杂度为 $O(n^3)$。本节讲解算法形式更为简单的 Floyd 算法,时间复杂度也为 $O(n^3)$。用户可以用 n 阶方阵序列来描述 Floyd 算法,其中 $D^{-1}[i][j]$表示从顶点出发不经过其他顶点直接到达顶点的路径长度,即 $D^{-1}[i][j]=$G. arcs$[i][j]$,$D^{(k)}[i][j]$表示从顶点 v_i到顶点 v_j的中间可能经过 $v_0,\cdots,v_k$,而不可能经过 $v_{k+1},\cdots,v_{n-1}$等顶点的最短路径长度,所以 $D^{(n-1)}[i][j]$是从顶点 v_i 到顶点 v_j 的最短路径长度,和路径长度序列相对应的是路径的 n 阶方阵序列 $\boldsymbol{p}^{(-1)},\boldsymbol{p}^{(0)},\boldsymbol{p}^{(1)},\cdots,\boldsymbol{p}^{(n-1)}$。

所以,Floyd 算法的基本操作可以概括为:

```
if D[i][k]+D[k][j]<D[i][j]:
    D[i][j]=D[i][k]+D[k][j]
    P[i][j]=P[i][k]+P[k][j]
```

其中,k 表示在路径中新增的顶点,i 为路径的源点,j 为路径的终点。

Floyd 算法构造最短路径的类用 Python 语言描述如下:

```
def Floyd(g):
    vNum = g.getVNum()
    D = [ [ sys.maxsize ] * vNum for i in range(vNum)]
    p = [ [ -1 ] * vNum for i in range(vNum)]
    for u in range(vNum):
        for v in range(vNum):
            D[u][v] = g.getArcs(u,v) if u!=v else 0
            if D[u][v]<sys.maxsize:
                p[u][v] = u
    for k in range(vNum):
        for i in range(vNum):
            for j in range(vNum):
                if D[i][j]>D[i][k]+D[k][j]:
                    D[i][j] = D[i][k] + D[k][j]
                    p[i][k] = i
                    p[i][j] = p[k][j]
```

```
    dis = {(g.getVex(u),g.getVex(v)):D[u][v] for u in range(vNum) for v in range(vNum)}
    return dis,p
def printFloydPath(g,p):
    vNum = g.getVNum()
    for u in range(vNum):
        for v in range(vNum):
            if u == v:
                print('%s->%s的最短路径为: %s' % (g.getVex(u),g.getVex(v),g.getVex(u)))
                continue
            flag = True
            path = [v]
            t = p[u][path[0]]
            while t!= u:
                if t ==-1:
                    flag = False
                    break
                path = [t] + path
                t = p[u][t]
            print('%s->%s的最短路径为:' % (g.getVex(u),g.getVex(v)),end = '')
            if flag:
                print(g.getVex(u),end = '')
                for node in path:
                    print('->%s' % g.getVex(node),end = '')
            print()
```

6.6 拓扑排序和关键路径

在生产实践中,几乎所有的工程都可以分解为若干个具有相对独立的子工程,称为"活动"。活动之间又通常受到一定条件的约束,即某些活动必须在另一些活动完成之后才能进行。可以使用有向图表示活动之间相互制约的关系,顶点表示活动,弧表示活动之间的优先关系,这种有向图称为顶点活动网(AOV)。若在 AOV 网中存在一条从顶点 u 到顶点 v 的弧,则活动 u 一定优先于活动 v 发生,否则活动 u、v 的发生顺序可以是任意的。

在 AOV 网中不允许出现环,否则某项活动的进行以其本身的完成作为先决条件,这是不允许发生的。判断有向网中是否存在环的方法是进行拓扑排序。

6.6.1 拓扑排序

对 AOV 网进行拓扑排序即构造一个包含图中所有顶点的拓扑有序序列,若在 AOV 网中存在一条从顶点 u 到顶点 v 的弧,则在拓扑有序序列中顶点 u 必须先于顶点 v,否则顶点 u、v 的顺序可以是任意的。AOV 网的拓扑有序序列并不唯一。若 AOV 网中存在环,则不可能将所有的顶点都纳入到拓扑有序序列中,因此可以用拓扑排序判断有向网中是否存在环。拓扑排序的主要步骤如下。

(1) 在 AOV 网中选择一个没有前驱的顶点并输出。

(2) 从 AOV 网中删除该顶点以及从它出发的弧。

(3) 重复步骤(1)和(2)直到 AOV 网为空,或者剩余子图中不存在没有前驱的顶点,此时说明 AOV 网中存在环。

整个拓扑排序可以分成求各个顶点的入度和一个拓扑序列的过程,具体算法描述如下。

【算法 6.16】 求各顶点的入度。

```
def findInDegree(g):
    indegree = [ 0 ] * g.getVNum()
    # 计算每个点的入度
    for u in range(g.getVNum()):
        v = g.firstAdj(u)
        while v!=-1:
            indegree[v] += 1
            v = g.nextAdj(u,v)
    return indegree
```

【算法 6.17】 计算 AOV 的一个拓扑序列,若存在返回拓扑序列,否则返回 None。

```
def topoSort(g):
    count = 0
    indegree = findInDegree(g)
    s = LinkStack()
    topo = []
    for i in range(g.getVNum()):
        # 入度为 0 的点入栈
        if indegree[i] == 0:
            s.push(i)
    while not s.isEmpty():
        u = s.pop()
        topo.append(u)
        count += 1
        # 对该点的每个邻接点的入度 - 1
        v = g.firstAdj(u)
        while v!=-1:
            indegree[v] -= 1
            if indegree[v] == 0:
                s.push(v)
            v = g.nextAdj(u,v)
    if count < g.getVNum():
        return None
    return [ g.getVex(u) for u in topo]
```

6.6.2 关键路径

若以弧表示活动,弧上的权值表示进行该项活动需要的时间,顶点表示事件,这种有向网称为边活动网络,简称为 AOE 网。弧指向事件表示该弧代表的活动已经完成,弧从事件出发表示该弧代表的活动开始进行,所以 AOE 网不允许环的存在。

AOE 网常用来表示工程的进行,表示工程开始事件的顶点的入度为 0,称为源点,表示工程结束事件的顶点的出度为 0,称为汇点,一个工程的 AOE 网应该是只有一个源点和一个汇点的有向无环图。由于 AOE 网中的某些活动可以并行进行,故完成整个工程的最短

时间即从源点到汇点的最长路径的长度，这条路径称为关键路径，构成关键路径的弧即为关键活动。

假设 V_0 为源点，V_{n-1} 为汇点，事件的发生时刻为 0 时刻。从 V_0 到 V_i 的最长路径叫做事件的最早发生时间，$e(i)$ 表示活动的最早开始时间，$l(i)$ 表示活动的最晚开始时间，指的是在不推迟整个工程的前提下，活动最晚必须开始的时间。当 $e(i)=l(i)$，称为关键活动。提前完成非关键活动并不能加快工程的进度，如果要缩短整个工期，必须首先找到关键路径，提高关键活动的工效。

根据事件的最早发生时间和最晚发生时间的定义可以采用下列步骤求得关键活动。

(1) 从源点出发，令 $\mathrm{ve}(0)=0$，其余各顶点的 $\mathrm{ve}(j)=\max(\mathrm{ve}(i)+|i,l|)$，其中，$T$ 是所有以第 j 个顶点为头的弧的集合。若得到的拓扑排序序列中顶点的个数小于网中的顶点个数 n，则说明网中有环，不能求出关键路径，算法结束。

(2) 从 Vn－1 汇点出发，令 $\mathrm{vl}(n-1)=\mathrm{ve}(n-1)$，按逆拓扑排序求其余各顶点允许的最晚开始时间为 $\mathrm{vl}(i)=\min(\mathrm{vl}(j)-|i,j|)$，其中，$S$ 是所有以第 j 个顶点为尾的弧的集合。

(3) 每一项活动 ai 的最早开始时间为 $e(i)=\mathrm{ve}(j)$，最晚开始时间为 $l(i)=\mathrm{vl}(j)-|i,j|$。若 ai 满足 $e(i)=l(i)$，则它是关键活动。

算法的具体描述如下。

【算法 6.18】 若拓扑序列存在，返回各顶点的最早发生时间 ve 与逆拓扑序列栈 t，否则返回 None，None。

```
def topoOrder(g):
    count = 0
    indegree = findInDegree(g)
    s = LinkStack()
    t = LinkStack() # 记录拓扑顺序
    ve = [ 0 ] * g.getVNum()
    for i in range(g.getVNum()):
        # 入度为 0 的点入栈
        if indegree[i] == 0:
            s.push(i)
    while not s.isEmpty():
        u = s.pop()
        t.push(u)
        count += 1
        # 对该点的每个邻接点的入度 - 1
        v = g.firstAdj(u)
        while v!= - 1:
            indegree[v] -= 1
            if indegree[v] == 0:
                s.push(v)
            # 更新最早发生时间
            if ve[u] + g.getArcs(u,v) > ve[v]:
                ve[v] = ve[u] + g.getArcs(u,v)
            v = g.nextAdj(u,v)
    if count < g.getVNum():
        return None,None
```

```
        return ve,t
```

【算法 6.19】 求各顶点的最晚发生时间并输出关键活动。

```
def criticalPath(g):
    ve,t = topoOrder(g)
    if ve is None:
        return None
    vl = [ ve[t.top.data] ] * g.getVNum()
    # 逆拓扑序求各顶点的 vl 值
    while not t.isEmpty():
        u = t.pop()
        v = g.firstAdj(u)
        while v!=-1:
            if vl[v]-g.getArcs(u,v)<vl[u]:
                vl[u] = vl[v]-g.getArcs(u,v)
            v = g.nextAdj(u,v)
    # 输出关键活动
    print("关键活动为:",end='')
    for i in range(g.getVNum()):
        if ve[i]==vl[i]:
            print(g.getVex(i),end=' ')
    print()
```

小　　结

(1) 图是一种数据元素间具有"多对多"关系的非线性数据结构,由顶点集 V 和边集 E 组成,记作 $G=(V,E)$。

(2) 图的常见存储结构有邻接矩阵、邻接表、十字链表 3 种。邻接矩阵是图,用二维数组存储;邻接表和十字链表是图的链式存储结构。

(3) 图的遍历是指从图的任意一个顶点出发对图的每个顶点访问且仅访问一次的过程。图的遍历方式分为两种,即广度优先搜索遍历和深度优先搜索遍历。

(4) 由广度优先遍历和深度优先遍历得到的生成树分别称为广度优先生成树和深度优先生成树。在一个网的所有生成树中权值总和最小的生成树称为最小代价生成树,简称为最小生成树,最小生成树不一定唯一。建立最小生成树的方法有 Kruskal 算法和 Prim 算法。

(5) 最短路径的求解问题主要分为两类,即求某个顶点到其余顶点的最短路径、求每一对顶点间的最短路径,可以分别使用 Dijkstra 算法和 Floyd 算法解决这两类问题。

(6) 用户可以使用有向图表示活动之间相互制约的关系,顶点表示活动,弧表示活动之间的优先关系,这种有向图称为顶点活动网(AOV)。若在 AOV 网中存在一条从顶点 u 到顶点 v 的弧,则活动 u 一定优先于活动 v 发生。

(7) 若以弧表示活动,弧上的权值表示进行该项活动需要的时间,顶点表示事件,这种有向网称为边活动网络,简称为 AOE 网。AOE 网络常用来表示工程的进行,一个工程的 AOE 网应该是只有一个源点和一个汇点的有向无环图。由于 AOE 网中的某些活动可以

并行进行，故完成整个工程的最短时间即从源点到汇点的最长路径的长度，这条路径称为关键路径，构成关键路径的弧即为关键活动。

习 题 6

一、选择题

1. 某无向图的邻接矩阵 $\boldsymbol{A}=\begin{bmatrix}0 & 1 & 0\\1 & 0 & 1\\0 & 1 & 0\end{bmatrix}$，可以看出该图共有(　　)个顶点。

A. 3　　B. 6

C. 9　　D. 以上答案均不正确

2. 无向图的邻接矩阵是一个(　　)，有向图的邻接矩阵是一个(　　)。

A. 上三角矩阵　　B. 下三角矩阵　　C. 对称矩阵　　D. 无规律

3. 下列命题正确的是(　　)。

A. 一个图的邻接矩阵表示是唯一的，邻接表表示也唯一

B. 一个图的邻接矩阵表示是唯一的，邻接表表示不唯一

C. 一个图的邻接矩阵表示是不唯一的，邻接表表示是唯一的

D. 一个图的邻接矩阵表示是不唯一的，邻接表表示也不唯一

4. 在一个具有 n 个顶点的有向完全图中包含有(　　)条边。

A. $n(n-1)/2$　　B. $n(n-1)$　　C. $n(n+1)/2$　　D. n^2

5. 一个具有 n 个顶点、k 条边的无向图是一个森林($n>k$)，则该森林中必有(　　)棵树。

A. k　　B. n　　C. $n-k$　　D. 1

6. 用深度优先遍历方法遍历一个有向无环图，并在深度优先遍历算法中按退栈次序打印出相应的顶点，则输出的顶点序列是(　　)。

A. 逆拓扑有序　　B. 拓扑有序

C. 无序　　D. 深度优先遍历序列

7. 关键路径是 AOE 网中(　　)。

A. 从源点到终点的最长路径　　B. 从源点到终点的最长路径

C. 最长的回路　　D. 最短的回路

二、填空题

1. 设无向图 G 中顶点数为 n，则图 G 最少有________条边、最多有________条边；若 G 为有向图，则最少有________条边、最多有________条边。

2. 任何连通图的连通分量只有一个，即________。

3. 图的存储结构主要有两种，分别是________和________。

4. 已知无向图 G 的顶点数为 n、边数为 e，其邻接表表示的空间复杂度为________。

5. 已知一个有向图的邻接矩阵表示，计算第 j 个顶点的入度的方法是________。

6. 有向图 G 用邻接矩阵 $\mathbf{A}[n][n]$ 存储,其第 i 行的所有元素之和等于顶点 i 的________。

7. 图的深度优先遍历类似于树的________遍历,它所用到的数据结构是________;图的广度优先遍历类似于树的________遍历,它所用到的数据结构是________。

8. 对于含有 n 个顶点、e 条边的连通图,利用 Prim 算法求最小生成树的时间复杂度为________,利用 Kruskal 算法求最小生成树的时间复杂度为________。

9. 如果一个有向图不存在________,则该图的全部顶点可以排列成一个拓扑序列。

10. 在一个有向图中若存在弧,则在其拓扑序列中顶点 v_i、v_j、v_k 的相对次序为________。

11. 在一个无向图中,所有顶点的度数之和等于所有边数的________倍。

12. n 个顶点的强连通图至少有________条边的形状是________。

13. 含 n 个顶点的连通图中的任意一条简单路径的长度不可能超过________。

14. 对于一个有 n 个顶点的无向图,若采用邻接矩阵存储,则该矩阵的大小是________。

15. 图的生成树________,n 个顶点的生成树有________条边。

16. G 是一个非连通无向图,共有 28 条边,则该图至少有________个顶点。

三、算法设计题

1. 设计一个算法,将一个无向图的邻接矩阵转换为邻接表。

2. 设计一个算法,计算出图中出度为零的顶点个数。

3. 以邻接表为存储结构,设计按深度优先遍历图的非递归算法。

4. 已知一个有向图的邻接表,编写算法建立其逆邻接表。

5. 设计一个算法,分别基于深度优先搜索和广度优先搜索编写算法,判断以邻接表存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径。

6. 图 6.11 所示为一个无向带权图,请分别按 Prim 算法和 Kruskal 算法求最小生成树。

7. 求图 6.12 中源点 v_1 到其他各顶点的最短路径。

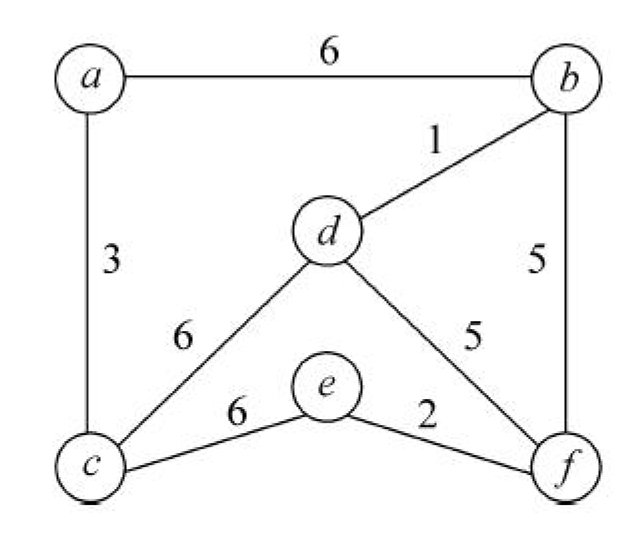


图 6.11 无向带权图

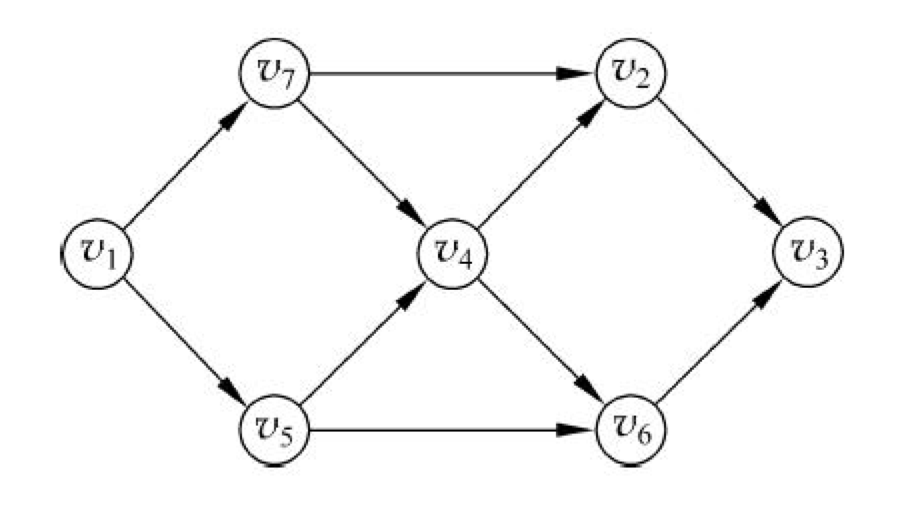


图 6.12 有向网图

第7章 排 序

7.1 排序概述

7.1.1 排序的基本概念

排序是指将一组数据按照关键字值的大小(递增或者递减)次序进行排列。排序是线性表、二叉树等数据结构的一种基本操作。作为排序依据的数据项叫关键字。关键字分为两种,一种关键字能唯一标识一条记录,叫主关键字;一种关键字标识多条记录,叫次关键字。可以指定一个数据元素的多个数据项分别作为关键字进行排序,显然排序结果不同。例如学号、班级、成绩等数据项均可以作为学生信息数据元素的关键字,按主关键字进行排序,结果唯一;按非主关键字进行排序,结果不唯一,如按班级排序,学生的次序不能确定,哪个在前后都有可能。

按照排序过程中所涉及的存储器的不同可将排序分为内部排序和外部排序两种类型。内部排序是指排序序列完全存放在内存中的排序过程;外部排序是指需要将数据元素存储在外部存储器上的排序过程。

排序又可分为稳定排序和不稳定排序。稳定排序是指在用某种排序算法依据关键字进行排序后具有相同关键字的数据元素的位置关系与排序前相同的排序过程,反之则为不稳定排序。

7.1.2 排序算法的性能评价

通常从时间复杂度和空间复杂度两个方面评价排序算法的性能。排序的时间复杂度主要用算法执行过程中的比较和移动次数来计算;排序的空间复杂度主要用外部存储空间的大小来计算。

排序往往处于软件的核心部分,经常被使用,所以其性能的优劣对软件质量的好坏起着重要的作用。

7.1.3 待排序的记录和顺序表的类描述

因为待排序的数据元素通常存储在顺序表中,所以本章中的排序算法都是以顺序表为基础进行设计的。

待排序的记录的类 Python 语言描述如下:

```
class RecordNode(object):
```

```
    def __init__(self,key,data):
        self.key = key          # 关键字
        self.data = data        # 数据元素的值
```

待排序的顺序表的类 Python 语言描述如下：

```
class SqList(object):
    def __init__(self,maxSize):
        self.maxSize = maxSize # 顺序表的最大存储空间
        self.list = [ None ] * self.maxSize # 待排序的记录集合
        self.len = 0 # 顺序表的长度

    def insert(self,i,x):
        # 在第 i 个位置之前插入记录 x
        if self.len == self.maxSize:
            raise Exception("顺序表已满")
        if i < 0 or i > self.len - 1:
            raise Exception("插入位置不合理")
        for j in range(self.len - 1,i, - 1):
            self.list[j] = self.list[j - 1]
        self.list[i] = x
        self.len += 1
```

7.2 插 入 排 序

7.2.1 直接插入排序

1. 直接插入算法的实现

直接插入排序是指将一条待排序的记录按照其关键字值的大小插入到已排序的记录序列中的正确位置，依次重复，直到全部记录都插入完成。其主要步骤如下。

(1) 将 list[i]存放在临时变量 p 中。

(2) 将 p 与 list[$i-1$]、list[$i-2$]、…、list[0]依次比较，若有 $p<$list[j]. key($j=i-1$、$i-2$、…、0)，则将 list[j]后移一个位置，直到 $p\geqslant$list[j]. key 为止。当 $p\geqslant$list[j]. key 时将 p 插入到 list[$j+1$]的位置。

(3) 令 $i=1$、2、3、…、$n-1$，重复步骤(1)、(2)、(3)。

假设一组待排序的记录的关键字序列为{2,45,36,72,34}，直接插入排序的过程如图 7.1 所示。

初始序列：[2] 45 36 72 34
p=1 : [2 45] 36 72 34
p=2 : [2 36 45] 72 34
p=3 : [2 36 45 72] 34
p=4 : [2 34 36 45 72]

图 7.1 直接插入的过程

【算法 7.1】 直接插入排序。

```
def insertSort(self):
    # 进行 len - 1 次扫描
    for i in range(1,self.len):
        p = self.list[i]
        # 比 list[i]大的元素后移
        j = i - 1
```

```
        while j>=0:
            if self.list[j].key>p.key:
                self.list[j+1] = self.list[j]
                j -= 1
            else:
                break
        self.list[j+1] = p # 插入
```

2. 算法性能分析

1）时间复杂度

有序表中逐个插入的操作进行了 $n-1$ 趟，每趟的插入操作的时间主要耗费在关键字的比较和数据元素的移动上。

在最好情况下待排列的顺序表已按关键字值有序，每趟排序比较一次，移动两次，总的比较和移动次数为 $3(n-1)$；在最坏情况下待排序的顺序表已按关键字值逆序排列，每趟比较 i 次，移动 $i+2$ 次，总的比较和移动次数为 $\sum_{i=1}^{n-1}(2i+2)=n^2+n$；在一般情况下，排序记录是随机序列，第 i 趟排序所需的比较和移动次数取平均值，约为 i 次，总的比较和移动次数为 $\sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}\approx\frac{n^2}{2}$，因此直接插入排序的时间复杂度为 $O(n^2)$。

2）空间复杂度

由于其仅使用了一个辅助存储单元 p，所以空间复杂度为 $O(1)$。

3）算法稳定性

在使用直接插入排序后具有相同关键字的数据元素的位置关系与排序前相同，因此直接插入排序是一种稳定的排序算法。

7.2.2 希尔排序

1. 希尔排序算法的实现

希尔排序是 D. L. Shell 在 1959 年提出的，又称缩小增量排序，是对直接插入排序的改进算法，其基本思想是分组的直接插入排序。

由直接插入排序算法分析可知，数据序列越接近有序则时间效率越高，当 n 较小时时间效率也较高。希尔排序正是针对这两点对直接插入排序算法进行改进。希尔排序算法的描述如下。

(1) 将一个数据元素序列分组，每组由若干个相隔一段距离的元素组成，在一个组内采用直接插入算法进行排序。

(2) 增量的初值通常为数据元素序列的一半，以后每趟增量减半，最后值为 1。随着增量逐渐减小，组数也减少，组内元素的个数增加，数据元素序列接近有序。

其主要步骤如下。

(1) 设定一个增量序列 $\{d_0, d_1, \cdots, 1\}$。

(2) 根据当前增量 d_i 将间隔为 d_i 的数据元素组成一个子表，共 d_i 个子表。

(3) 对各子表中的数据元素进行直接插入排序。

(4) 重复步骤(2)、(3)，直到进行完 $d_i=1$，此时序列已按关键字值排序。

假设一组待排序的记录的关键字序列为{2,18,23,56,78,70,45,36,72,34},增量分别取 5、3、1,则希尔排序的过程如图 7.2 所示。

初始序列:2 18 23 56 78 70 45 36 72 34
d=5
i=1 2 18 23 56 34 70 45 36 72 78
d=3
i=2 2 18 23 45 34 70 56 36 72 78
d=1
i=3 2 18 23 34 36 45 56 70 72 78

图 7.2 希尔排序的过程

【算法 7.2】 希尔排序。

```
def shellSort(self,d):
    for k in d:
        # 在增量内进行直接插入排序
        j = k
        while j < self.len:
            p = self.list[j]
            m = j
            while m >= k:
                if self.list[m-k].key > p.key:
                    self.list[m] = self.list[m-k]
                    m = m-k
                else:
                    break
            self.list[m] = p
            j += 1
```

2. 算法性能分析

1) 时间复杂度

希尔排序的关键字比较次数和数据元素的移动次数取决于增量的选择,目前还没有更好的选取增量序列的方法。Hibbard 提出了一种增量序列{$2^k-1,2^{k-1}-1,\cdots,7,3,1$},可以使时间复杂度达到 $O(n^{3/2})$。需要注意的是,在增量序列中应没有除 1 以外的公因子,并且最后一个增量值必须为 1。

2) 空间复杂度

希尔排序仍只使用了一个额外的存储空间 p,其空间复杂度为 $O(1)$。

3) 算法稳定性

希尔排序算法在比较过程中会错过关键字相等的数据元素的比较,算法不能控制稳定,因此希尔排序是一种不稳定的排序算法。

7.3 交换排序

基于交换的排序算法主要有两种,即冒泡排序和快速排序。

7.3.1 冒泡排序

1. 冒泡排序算法的实现

冒泡排序是两两比较待排序记录的关键字,如果次序相反则交换两个记录的位置,直到序列中的所有记录有序。若按升序排序,每趟将数据元素序列中的最大元素交换到最后的位置,就像气泡从水里冒出一样。其主要步骤如下。

(1) 设交换次数 $k=1$。

(2) 在常数为 n 的序列{$a[0]$,$a[1]$,…,$a[n-1]$}中从头到尾比较 $a[i]$和 $a[i+1]$,若 $a[i]$.key>$a[i+1]$.key,则交换两个元素的位置,其中,$0\leqslant i<n-i$。

(3) k 增加 1。

(4) 重复步骤(2)、(3),直到 $k=n-1$ 或者步骤(2)中未发生交换为止。

假设一组待排序的记录的关键字序列为{2,23,18,56,78,70,45,36,72,34},冒泡排序的过程如图 7.3 所示。

初始序列: 2 23 18 56 78 70 45 36 72 34
第一趟: 2 18 23 56 70 45 36 72 34 78
第二趟: 2 18 23 56 45 36 70 34 72 78
第三趟: 2 18 23 45 36 56 34 70 72 78
第四趟: 2 18 23 36 45 34 56 70 72 78
第五趟: 2 18 23 36 34 45 56 70 72 78
第六趟: 2 18 23 34 36 45 56 70 72 78

图 7.3 冒泡排序的过程

【算法 7.3】 冒泡排序。

```
def bubbleSort(self):
    flag = True
    i = 1
    while i < self.len and flag:
        flag = False
        for j in range(self.len - i):
            if self.list[j + 1].key < self.list[j].key:
                p = self.list[j]
                self.list[j] = self.list[j + 1]
                self.list[j + 1] = p
                flag = True
        i += 1
```

2. 算法性能分析

1) 时间复杂度

在最好情况下排序表已经有序,只进行一趟冒泡排序,在这次操作中发生了 $n-2$ 次的比较;在最坏情况下排序表逆序,需要进行 $n-1$ 趟冒泡排序,在第 i 趟排序中比较次数为 $n-i$、移动次数为 $3(n-i)$,总的比较和移动次数为 $\sum_{i=1}^{n-1}4(n-i)=2(n^2-n)$;在一般情况下排序记录是随机序列,冒泡排序的时间复杂度为 $O(n^2)$。

2) 空间复杂度

冒泡排序仅用了一个辅助存储单元 p,所以其空间复杂度为 $O(1)$。

3) 算法稳定性

冒泡排序是一种稳定的排序算法。

7.3.2 快速排序

1. 快速排序算法的实现

快速排序是一种分区交换排序算法,是冒泡排序的改进,其采用了分治策略,将问题划分成若干个规模更小但和原问题相似的子问题,然后用递归方法解决这些子问题,最终将它们组合成原问题的解。

快速排序将要排序的序列分成独立的两个部分,其中一部分的关键字值都比另一部分的关键字值大,然后分别对这两个部分进行快速排序,排序过程递归进行,整个序列最终达到有序。

独立的两个部分的划分方法为在序列中任意选取一条记录,然后将所有关键字值比它大的记录放到它的后面,将所有关键字值比它小的记录放到它的前面。这条记录叫支点。

其主要步骤如下。

(1) 设置两个变量 low、high,分别表示待排序序列的起始下标和终止下标。

(2) 设置变量 p=list[low]。

(3) 从下标为 high 的位置从后向前依次搜索,当找到第 1 个比 p 的关键字值小的记录时将该数据移动到下标为 low 的位置上,low 加 1。

(4) 从下标为 low 的位置从前向后依次搜索,当找到第 1 个比 p 的关键字值大的记录时将该数据移动到下标为 high 的位置上,high 减 1。

(5) 重复步骤(3)和(4),直到 high=low 为止。

(6) list[low]=p。

假设一组待排序的记录的关键字序列为{45,53,18,36,72,30,48,93,15,36},以排序码 45 进行第一次划分的过程如图 7.4 所示。

```
[45  53  18  36  72  30  48  93  15  36]   移动比较
 ↑low                                ↑high
[36  53  18  36  72  30  48  93  15  36]   list[low]=list[high],low++
     ↑low                            ↑high
[36  53  18  36  72  30  48  93  15  36]   移动比较
     ↑low                            ↑high
[36  53  18  36  72  30  48  93  15  53]   list[high]=list[low],high--
     ↑low                        ↑high
[36  53  18  36  72  30  48  93  15  53]   移动比较
     ↑low                        ↑high
[36  15  18  36  72  30  48  93  15  53]   list[low]=list[high],low++
[36  15  18  36  72  30  48  93  15  53]   移动比较
[36  15  18  36  72  30  48  93  72  53]   list[high]=list[low],high--
[36  15  18  36  72  30  48  93  72  53]   移动比较
[36  15  18  36  30  30  48  93  72  53]   list[low]=list[high],low++
[36  15  18  36  30  45  48  93  72  53]   list[low]=p
[36  15  18  36  30] 45 [48  93  72  53]   完成一次划分
```

图 7.4　进行第一次划分的过程

快速排序的过程如图 7.5 所示。

```
[45  53  18  36  72  30  48  93  15  36]
[36  15  18  36  30] 45 [48  93  72  53]
[30  15  18] 36 [36] 45  48 [93  72  53]
[18  15] 30  36  36  45  48 [53  72] 93
[15] 18  30  36  36  45  48  53 [72] 93
 15  18  30  36  36  45  48  53  72  93
```

图 7.5　快速排序的过程

【算法 7.4】 快速排序。

```
def qSort(self,low,high):
    if low < high:
        p = self.Partition(low,high)
        self.qSort(low,p-1)
        self.qSort(p+1,high)

def Partition(self,low,high):
    p = self.list[low]
    while low < high:
        while low < high and self.list[high].key > p.key:
            high -= 1
        if low < high:
            self.list[low] = self.list[high]
            low += 1
        while low < high and self.list[low].key < p.key:
            low += 1
        if low < high:
            self.list[high] = self.list[low]
            high -= 1
    self.list[low] = p
    return low
```

2. 算法性能分析

1）时间复杂度

快速排序的执行时间与数据元素序列的初始排列以及基准值的选取有关。在最坏情况下待排序序列基本有序，每次划分只能得到一个子序列，等同于冒泡排序，时间复杂度为 $O(n^2)$；在一般情况下，对于具有 n 条记录的序列来说，一次划分需要进行 n 次关键字的比较，其时间复杂度为 $O(n)$，设 $T(n)$ 为对其进行快速排序所需要的时间，可得：

$$
\begin{aligned}
T(n) &\leqslant cn + 2T\left(\frac{n}{2}\right) \\
&\leqslant cn + 2\left(\frac{cn}{2} + 2T\left(\frac{n}{4}\right)\right) = 2cn + 4T\left(\frac{n}{4}\right) \\
&\leqslant 2cn + 4\left(\frac{cn}{4} + 2T\left(\frac{n}{8}\right)\right) = 3cn + 8T\left(\frac{n}{8}\right) \\
&\vdots \\
&\leqslant cn\log_2 n + nT(1) = O(n\log_2 n)
\end{aligned}
$$

所以快速排序是内部排序中速度最快的，其时间复杂度为 $O(n\log_2 n)$。

快速排序的基准值的选择有许多方法，可以选取序列的中间值等，但由于数据元素序列的初始排列是随机的，不管如何选择基准值总会存在最坏情况。总之，当 n 较大并且数据元素序列随机排列时快速排序是快速的；当 n 很小或者基准值选取不合适时，快速排序较慢。

2）空间复杂度

快速排序需要额外存储空间栈来实现递归，递归调用的指针的参数都要存放到栈中。快速排序的递归过程可用递归树来表示。在最坏情况下树为单枝树，高度为 $O(n)$，其空间复杂度为 $O(n)$。若划分较为均匀，二叉树的高度为 $O(\log 2n)$，其空间复杂度也为 $O(\log 2n)$。

3) 算法稳定性

快速排序是一种不稳定的排序算法。

【例 7.1】 对 n 个元素组成的顺序表进行快速排序时所需进行的比较次数与这 n 个元素的初始排序有关。问：

(1) 当 $n=7$ 时在最好情况下需进行多少次比较？请说明理由。

(2) 当 $n=7$ 时给出一个最好情况下的初始排序的实例。

(3) 当 $n=7$ 时在最坏情况下需进行多少次比较？请说明理由。

(4) 当 $n=7$ 时给出一个最坏情况下的初始排序的实例。

解：

(1) 在最好情况下每次划分能得到两个长度相等的子文件。假设文件的长度 $n=2k-1$，那么第一遍划分得到两个长度均为 $n/2$ 的子文件，第二遍划分得到 4 个长度均为 $n/4$ 的子文件，以此类推，总共进行 $k=\log_2(n+1)$ 遍划分，各子文件的长度均为 1，排序完毕。当 $n=7$ 时，$k=3$，在最好情况下第一遍需比较 6 次，第二遍分别对两个子文件(长度均为 3，$k=2$)进行排序，各需两次，共 10 次即可。

(2) 在最好情况下快速排序的原始序列实例为{4,1,3,2,6,5,7}。

(3) 在最坏情况下若每次用来划分的记录的关键字具有最大(或最小)值，那么只能得到左(或右)子文件，其长度比原长度少 1。因此，若原文件中的记录按关键字递减次序排列，而要求排序后按递增次序排列，快速排序的效率与冒泡排序相同，所以当 $n=7$ 时最坏情况下的比较次数为 21 次。

(4) 在最坏情况下快速排序的初始序列实例为{7,6,5,4,3,2,1}要求按递增排序。

7.4 选择排序

7.4.1 直接选择排序

1. 直接选择排序算法的实现

直接选择排序是从序列中选择关键字值最小的记录进行放置，直到整个序列中的所有记录都选完为止。直接选择排序在第一次选择中从 n 个记录中选出关键字值最小的记录与第一个记录交换，在第二次选择中从 $n-1$ 个元素中选取关键字值最小的记录与第二个记录交换，以此类推，在第 i 次选择中从 $n-i+1$ 个元素中选取关键字值最小的记录和第 i 个记录交换，直到整个序列按关键字值有序时停止。

其主要步骤如下。

(1) 令 $i=0$。

(2) 在无序序列$\{a_i, a_{i+1}, \cdots, a_{n-1}\}$中选出关键字值最小的记录 $a_{\min}$。

(3) $a_{\min}$ 与 a_i 交换位置，i 加 1。

(4) 重复步骤(2)和(3)，直到 $i=n-2$ 时停止。

假设一组待排序的记录的关键字序列为{36,23,18,56,78,70,45,2}，直接选择排序的过程如图 7.6 所示。

初始数据：36 23 18 56 78 70 45 2

第一趟： [2] 23 18 56 78 70 45 36

第二趟： [2 18] 23 56 78 70 45 36

第三趟： [2 18 23] 56 78 70 45 36

第四趟： [2 18 23 36] 78 70 45 56

第五趟： [2 18 23 36 45] 70 78 56

第六趟： [2 18 23 36 45 56] 78 70

第七趟： [2 18 23 36 45 56 70 78]

图 7.6 直接选择排序过程

【算法 7.5】 直接选择排序。

```
def selectSort(self):
    for i in range(self.len - 1): # 进行 n - 1 趟选择
        tmp = i
        for j in range(i,self.len): # 寻找关键字值最小的记录的位置
            if self.list[j].key < self.list[tmp].key:
                tmp = j
        # 交换位置
        p = self.list[i]
        self.list[i] = self.list[tmp]
        self.list[tmp] = p
```

2. 算法性能分析

1）时间复杂度

直接选择排序的比较次数与数据元素序列的初始排列无关，移动次数与初始排列有关。直接选择排序的移动次数较少，最好情况为序列有序，移动 0 次，最坏情况为序列逆序，移动 $3(n-1)$次。其比较次数较多，进行了 $n-1$ 趟选择，每趟需要进行 $n-i-1$ 次比较，所以总的比较次数为 $\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}$，所以直接选择排序的时间复杂度为 $O(n^2)$。

2）空间复杂度

直接选择排序过程用了一个额外的存储单元 p，所以其空间复杂度为 $O(1)$。

3）算法稳定性

直接选择排序是一种不稳定的排序算法。

7.4.2 堆排序

1. 堆的定义

假设有 n 个记录关键字的序列为 $\{k_0, k_1, \cdots, k_{n-1}\}$，当且仅当满足下面的条件时称为堆。

$$\begin{cases} k_i \leqslant k_{2i+1}, 2i+1<n \\ k_i \leqslant k_{2i+2}, 2i+2<n \end{cases}$$

或

$$\begin{cases} k_i \geqslant k_{2i+1}, 2i+1 < n \\ k_i \geqslant k_{2i+2}, 2i+2 < n \end{cases}$$

前者称为小顶堆,后者称为大顶堆。

直接选择排序算法有以下两个缺点。

(1) 选择最小值效率低,必须遍历子序列,比较了所有元素后才能选出最小值。

(2) 每趟将最小值交换到前面,其余元素原地不动,下一趟没有利用前一趟的比较结果,需要重复进行数据元素关键字值的比较,效率较低。

堆排序是利用完全二叉树特性的一种选择排序。虽然堆中的记录无序,但在小顶堆中堆顶记录的关键字值最小,在大顶堆中堆顶记录的关键字值最大,因此堆排序是首先将 n 条记录按关键字值的大小排成堆,将堆顶元素与第 $n-1$ 个元素交换位置并输出,再将前 $n-1$ 个记录排成堆,将堆顶元素与第 $n-2$ 个元素交换并输出,以此类推,即可得到一个按关键字值进行排序的有序序列。

2. 用筛选法调整堆

在进行堆排序的过程中,当堆顶元素和堆中的最后一个元素交换位置后根节点和其子节点的关键字值不再满足堆的定义,需要进行调整。

用筛选法调整堆是将根节点和其左、右孩子节点的关键字值进行比较,其与具有较小关键字值的孩子节点进行交换。被交换的孩子节点所在的子树可能不再满足堆的定义,重复对不满足堆定义的子树进行交换操作,直到堆被建成。

调整堆的主要步骤如下。

(1) 设置变量 i 为需要调整的序列的最小下标 low,设置变量 $j=2i+1$,设置变量 $p=$ list[i]。

(2) 当 $j<$high-1 时,若 list[j].key$>$list[$j+1$].key,j 加 1。

(3) 若 $p>$list[j].key,则 list[i]$=$list[j],$i=j$,$j=2i=1$。

(4) 重复步骤(2)、(3),直到 $j\geqslant$high。

(5) list[i]$=p$。

【算法 7.6】 用筛选法调整堆。

```
def sift(self,low,high):
    i = low
    j = 2 * i + 1
    p = self.list[i]
    while j < high:
        if j < high - 1 and self.list[j].key > self.list[j + 1].key:
            # 比较左右孩子的关键字大小
            j += 1
        if p.key > self.list[j].key:
            # 交换父节点和子节点并相加进行筛选
            self.list[i] = self.list[j]
            i = j
            j = 2 * i + 1
        else:
```

```
                # 退出循环
                j = high + 1
        self.list[i] = p
```

3. 堆排序

堆排序的主要步骤如下。

(1) 将待排序序列建成一棵完全二叉树。

(2) 将完全二叉树建堆。

(3) 输出堆顶元素并用筛选法调整堆,直到二叉树只剩下一个节点。

为一个序列建堆的过程就是对完全二叉树进行从下往上反复筛选的过程。筛选从最后一个非叶节点开始向上进行,直到对根节点进行筛选,堆被建成。

【算法 7.7】 堆排序。

```
def heapSort(self):
    for i in range(self.len//2 - 1, - 1, - 1):       # 创建堆
        self.sift(i,self.len - 1)
    for i in range(self.len - 1,0, - 1):             # 用筛选法调整堆
        p = self.list[0]
        self.list[0] = self.list[i]
        self.list[i] = p
        self.sift(0,i - 1)
```

4. 算法性能分析

1) 时间复杂度

假设在堆排序过程中产生的二叉树的树高为 k,则 $k=\lfloor \log_2 n \rfloor+1$,一次筛选过程,关键字的比较次数最多为 $2(k-1)$ 次,交换次数最多为 k 次,所以堆排序总的比较次数不超过 $2(\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-1) \rfloor+\cdots+\lfloor \log_2 2 \rfloor)<2n\log_2 n$。建初始堆的比较次数不超过 $4n$ 次,所以在最坏情况下堆排序算法的时间复杂度为 $O(n\log_2 n)$。

2) 空间复杂度

堆排序需要一个额外的存储空间 p,其空间复杂度为 $O(1)$。

3) 算法稳定性

堆排序算法是不稳定的排序算法。

【例 7.2】 判断下面的每个节点序列是否表示一个堆,如果不是堆,请把它调整成堆。

(1) 100,90,80,60,85,75,20,25,10,70,65,50

(2) 100,70,50,20,90,75,60,25,10,85,65,80

解:

(1) 是堆。

(2) 不是堆。调成大堆:100,90,80,25,85,75,60,20,10,70,65,50

7.5 归并排序

归并排序是指将两个或者两个以上的有序表合并成一个新的有序表,其中有序表个数为 2 的归并排序叫二路归并排序,其他的叫多路归并排序。

1. 两个相邻有序序列归并

两个有序序列分别存放在一维数组的 a[i..k]和 a[k+1..j]中,设置数组 order[]存放合并后的有序序列。归并排序的主要步骤如下。

(1) 比较两个有序序列的第 1 个记录的关键字值的大小,将关键字值较小的记录放入数组 order[]中。

(2) 对剩余的序列重复步骤(1)的过程,直到所有的记录都放入了有序数组 order[]。

假设一组待排序的记录的关键字序列为{45,53,18,36,72,30,48,93,15,36},归并排序的过程如图 7.7 所示。

[45] [53] [18] [36] [72] [30] [48] [93] [15] [36]

[45 53] [18 36] [72 30] [48 93] [15 36]

[18 36 45 53] [30 48 72 93] [15 36]

[18 30 36 45 48 53 72 93] [15 36]

[15 18 30 36 36 45 48 53 72 93]

图 7.7 归并排序的过程

【算法 7.8】 两个相邻有序序列归并。

```
def merge(self,order,a,i,k,j):
    t = i
    m = i
    n = k + 1
    while m<=k and n<=j:
        # 将具有较小关键字值的元素放入 order[]
        if a[m].key<=a[n].key:
            order[t] = a[m]
            t += 1
            m += 1
        else:
            order[t] = a[n]
            t += 1
            n += 1
    while m<=k:
        order[t] = a[m]
        t += 1
        m += 1
    while n<=j:
        order[t] = a[n]
        t += 1
        n += 1
```

2. 一趟归并排序

一趟归并排序过程是指将待排序中的有序序列两两合并的过程,合并结果仍放在数组 order[]中。

【算法 7.9】 一趟归并排序算法。

```
def mergepass(self,order,a,s,n):
    p = 0
    while p+2*s-1<=n-1: # 两两归并长度均为 s 的有序表
        self.merge(order,a,p,p+s-1,p+2*s-1)
        p = p+2*s
    if p+s-1<n-1: # 归并长度不等的有序表
        self.merge(order,a,p,p+s-1,n-1)
    else: # 将一个有序表中的元素放入 order[]中
        for i in range(p,n):
            order[i] = a[i]
```

3. 二路归并排序

【算法 7.10】 二路归并排序。

```
def mergeSort(self):
    s = 1 # 已排序的子序列的长度,初始值为 1
    order = [ None ] * self.len
    while s<self.len: # 归并过程
        self.mergepass(order,self.list,s,self.len)
        s = s * 2
        self.mergepass(self.list,order,s,self.len)
        s = s * 2
```

算法性能分析：

1）时间复杂度

二路归并排序算法的时间复杂度等于归并的趟数与每一趟时间复杂度的乘积。归并的趟数为 $\log_2 n$，每一趟归并的移动次数为数组中记录的个数 n，比较次数一定不大于移动次数，所以每一趟归并的时间复杂度为 $O(n)$，故二路归并排序算法的时间复杂度为 $O(n\log_2 n)$。

2）空间复杂度

二路归并排序算法需要使用一个与待排序序列等长的数组作为额外存储空间存放中间结果，所以其空间复杂度为 $O(n)$。

3）算法稳定性

二路归并排序算法是一种稳定的排序算法。

【例 7.3】 设待排序的关键字序列为{15,21,6,30,23,6',20,17}，试分别写出使用以下排序方法每趟排序后的结果。并说明做了多少次比较。

（1）直接插入排序。

（2）希尔排序（增量为 5、2、1）。

（3）冒泡排序。

（4）快速排序。

（5）直接选择排序。

（6）堆排序。

（7）二路归并排序。

解：

(1) 直接插入排序。

初始关键字序列：15,21,6,30,23,6',20,17

第一趟直接插入排序：【15,21】

第二趟直接插入排序：【6,15,21】

第三趟直接插入排序：【6,15,21,30】

第四趟直接插入排序：【6,15,21,23,30】

第五趟直接插入排序：【6,6',15,21,23,30】

第六趟直接插入排序：【6,6',15,20,21,23,30】

第七趟直接插入排序：【6,6',15,17,20,21,23,30】

(2) 希尔排序(增量为 5、2、1)。

初始关键字序列：15,21,6,30,23,6',20,17

第一趟希尔排序：6',20,6,30,23,15,21,17

第二趟希尔排序：6',15,6,17,21,20,23,30

第三趟希尔排序：6',6,15,17,20,21,23,30

(3) 冒泡排序。

初始关键字序列：15,21,6,30,23,6',20,17

第一趟冒泡排序：15,6,21,23,6',20,17,30

第二趟冒泡排序：6,15,21,6',20,17,23,30

第三趟冒泡排序：6,15,6',20,17,21,23,30

第四趟冒泡排序：6,6',15,17,20,21,30,23

第五趟冒泡排序：6,6',15,17,20,21,23,30

(4) 快速排序。

初始关键字序列：15,21,6,30,23,6',20,17

第一趟快速排序：【6',6】15【30,23,21,20,17】

第二趟快速排序：6',6, 15【17,23,21,20】30

第三趟快速排序：6',6, 15,17【23,21,20】30

第四趟快速排序：6',6, 15,17【20,21】23,30

第五趟快速排序：6,6',15,17,20,21,23,30

(5) 直接选择排序。

初始关键字序列：15,21,6,30,23,6',20,17

第一趟直接选择排序：6,21,15,30,23,6',20,17

第二趟直接选择排序：6,6',15,30,23,21,20,17

第三趟直接选择排序：6,6',15,30,23,21,20,17

第四趟直接选择排序：6,6',15,17,23,21,20,30

第五趟直接选择排序：6,6',15,17,20,21,23,30

第六趟直接选择排序：6,6',15,17,20,21,23,30

第七趟直接选择排序：6,6',15,17,20,21,23,30

(6) 堆排序。

初始关键字序列：15,21,6,30,23,6',20,17

初始堆：6,17,6',21,23,15,20,30

第一次调堆：6',17,15, 21,23,30,20,【6】

第二次调堆：15,17,20,21,23,30,【6',6】

第三次调堆：17,21,20,30,23,【15,6',6】

第四次调堆：20,21,23,30,【17,15,6',6】

第五次调堆：21,30,23,【20,17,15,6',6】

第六次调堆：23,30,【21,20,17,15,6',6】

第七次调堆：30,【23,21,20,17,15,6',6】

堆排序结果调堆：【30,23,21,20,17,15,6',6】

(7) 二路归并排序。

初始关键字序列：15,21,6,30,23,6',20,17

二路归并排序结果：15,17,20,21,23,30,6',6

各类排序方法的平均时间复杂度、最坏情况时间复杂度和空间复杂度以及稳定性情况如表 7.1 所示。

表 7.1 排序方法比较

排序方法	平均时间	最坏情况	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
希尔排序	$O(n^{3/2})$	$O(n^{3/2})$	$O(1)$	不稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
二路归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定

小　　结

(1) 排序是指将一组数据按照关键字值的大小(递增或者递减)次序进行排列。按照排序过程中所涉及的存储器的不同可将排序分为内部排序和外部排序两种类型。排序又可分为稳定排序和不稳定排序。

(2) 常用的内部排序算法有插入排序、交换排序、选择排序、归并排序。

(3) 插入排序算法有两种,直接插入排序算法是将一条待排序的记录按照其关键字值的大小插入到已排序的记录序列中的正确位置,以此重复,直到全部记录都插入完成；希尔排序是分组的直接插入排序。

(4) 在交换排序中,冒泡排序是两两比较待排序记录的关键字,如次序相反则交换两个记录的位置,直到序列中的所有记录有序；快速排序是将要排序的序列分成独立的两个部分,其中一部分的关键字值都比另一部分的关键字值大,然后分别对这两个部分进行快速

排序。

(5) 在选择排序中,直接选择排序是从序列中选择关键字值最小的记录进行放置,直到整个序列中的所有记录都选完为止;堆排序是将 n 条记录按关键字值的大小排成堆,将堆顶元素与第 $n-1$ 个元素交换位置并输出,以此类推,即可得到有序序列。

(6) 归并排序是指将两个或者两个以上的有序表合并成一个新的有序表,其中有序表个数为 2 的归并排序叫二路归并排序,其他的叫多路归并排序。

习　题　7

一、选择题

1. 在下列内部排序算法中:

(1) 其比较次数与序列的初始状态无关的算法是(　　)。

(2) 不稳定的排序算法是(　　)。

(3) 在初始序列已基本有序(除去 n 个元素中的某 k 个元素后即呈有序,$k \gg n$)的情况下排序效率最高的算法是(　　)。

(4) 排序的平均时间复杂度为 $O(n\log_2 n)$的算法是(　　),为 $O(n^2)$的算法是(　　)。

A. 快速排序　　B. 直接插入排序
C. 二路归并排序　　D. 简单选择排序
E. 冒泡排序　　F. 堆排序

2. 比较次数与排序的初始状态无关的排序方法是(　　)。

A. 直接插入排序　　B. 起泡排序　　C. 快速排序　　D. 简单选择排序

3. 对一组数据{84,47,25,15,21}排序,数据的排列次序在排序过程中的变化为

(1) {84,47,25,15,21}　　(2) {15,47,25,84,21}
(3) {15,21,25,84,47}　　(4) {15,21,25,47,84}

则采用的排序是(　　)。

A. 选择　　B. 冒泡　　C. 快速　　D. 插入

4. 下列排序算法中(　　)排序在一趟结束后不一定能选出一个元素放在其最终位置上。

A. 选择　　B. 冒泡　　C. 归并　　D. 堆

5. 一组记录的关键码为{46,79,56,38,40,84},则利用快速排序的方法以第一个记录为基准得到的一次划分结果为(　　)。

A. {38,40,46,56,79,84}　　B. {40,38,46,79,56,84}
C. {40,38,46,56,79,84}　　D. {40,38,46,84,56,79}

6. 在下列排序算法中,在待排序数据已有序时花费的时间反而最多的是(　　)排序。

A. 冒泡　　B. 希尔　　C. 快速　　D. 堆

7. 就平均性能而言,目前最好的内排序方法是(　　)排序法。

A. 冒泡　　B. 希尔插入　　C. 交换　　D. 快速

8. 下列排序算法中，占用辅助空间最多的是(　　)。

A. 归并排序　　B. 快速排序　　C. 希尔排序　　D. 堆排序

9. 若用冒泡排序法对序列{10,14,26,29,41,52}从大到小排序，需要进行(　　)次比较。

A. 3　　B. 10　　C. 15　　D. 25

10. 快速排序法在(　　)情况下最不利于发挥其长处。

A. 要排序的数据量太大　　B. 要排序的数据中含有多个相同值

C. 要排序的数据个数为奇数　　D. 要排序的数据已基本有序

11. 在下列 4 个序列中(　　)是堆。

A. 75,65,30,15,25,45,20,10　　B. 75,65,45,10,30,25,20,15

C. 75,45,65,30,15,25,20,10　　D. 75,45,65,10,25,30,20,15

12. 有一组数据(15,9,7,8,20,−1,7,4)，用堆排序的筛选方法建立的初始堆为(　　)。

A. −1,4,8,9,20,7,15,7　　B. −1,7,15,7,4,8,20,9

C. −1,4,7,8,20,15,7,9　　D. A、B、C 均不对

二、填空题

1. 在索引顺序表中首先查找________，然后查找相应的________，其平均查找长度等于________。

2. 若待排序的序列中存在多个记录具有相同的键值，经过排序这些记录的相对次序仍然保持不变，则称这种排序方法是________的，否则称为________的。

3. 按照排序过程涉及的存储设备的不同排序可分为________排序和________排序。

4. 对 n 个记录的表 $r[1..n]$ 进行简单选择排序所需进行的关键字间的比较次数为________。

三、算法设计题

1. 一个线性表中的元素为正整数或负整数，设计算法将正整数和负整数分开，使线性表的前一半为负整数、后一半为正整数，不要求对这些元素排序，但要求尽量减少比较次数。

2. 已知$\{k_1,k_2,\cdots,k_n\}$是堆，试写一算法将$\{k_1,k_2,\cdots,k_n,k_{n+1}\}$调整为堆。

3. 给定 n 个记录的有序序列 $A[n]$ 和 m 个记录的有序序列 $B[m]$，将它们归并为一个有序序列，存放在 $C[m+n]$ 中，试写出这一算法。

4. 编写一个算法，在基于单链表表示的关键字序列上进行简单选择排序。

5. 设单链表的头节点指针为 L、节点数据为整型，试写出对链表 L 按“直接插入方法”排序的算法。

6. 试设计一个双向冒泡排序算法，即在排序过程中交替改变扫描方向。

7. 写出快速排序的非递归算法。

第8章　查　找

8.1　查找的基本概念

8.1.1　什么是查找

查找是数据结构的一种基本操作，查找的效率决定了计算机某些应用系统的效率。查找算法依赖于数据机构，不同的数据结构需要采用不同的查找算法，因此如何有效地组织数据以及如何根据数据结构的特点快速、高效地获得查找结果是数据处理的核心问题。

查找就是在由一组记录组成的集合中寻找属性值符合特定条件的数据元素。若集合中存在符合条件的记录，则查找成功，否则查找失败。查找条件由包含指定关键字的数据元素给出。

根据不同的应用需求，查找结果有以下表示形式。

(1) 如果判断数据结构是否包含某个特定元素，则查找结果为是、否两个状态。

(2) 如果根据关键字查找以获得特定元素的其他属性，则查找结果为特定数据元素。

(3) 如果数据结构中含有多个关键字值相同的数据元素，需要确定返回首次出现的元素或者是返回数据元素集合。

(4) 如果查找不成功，返回相应的信息。

8.1.2　查找表

查找表是一种以同一类型的记录构成的集合为逻辑结构、以查找为核心运算的灵活的数据结构。在实现查找表时要根据实际情况按照查找的具体要求组织查找表，从而实现高效率的查找。

在查找表中常做的操作有建表、查找、读表、插入和删除。查找表分为静态查找表和动态查找表两种。静态查找表是指对表的操作不包括对表的修改的表；动态查找表是指对表的操作包括对表中的记录进行插入和删除的表。

8.1.3　平均查找长度

查找的主要操作是关键字的比较，所以衡量一个查找算法效率优劣的标准是比较次数的期望值。给定值与关键字值的比较次数的期望值也称为平均查找长度(Average Search Length)，记为 ASL。

对于一个含有 n 个记录的查找表，查找成功时的平均查找长度如下：

$$\mathrm{ASL} = \sum_{i=0}^{n-1} p_i c_i$$

其中,p_i是查找第 i 条记录的概率,c_i 是查找第 i 条记录时关键字值和给定值比较的次数。

8.2 静态表查找

静态查找表是指对表的操作不包括对表的修改的表,可以用顺序表或线性链表表示。本节中只讨论顺序表上查找的实现方法,分为顺序查找、二分查找和分块查找 3 种。此外,假设关键字值为 int 类型,采用第 7 章实现的顺序表类 SeqList 和记录节点类 RecordNode。

8.2.1 顺序查找

1. 顺序查找算法的实现

顺序查找是指从顺序表的一端开始依次将每一个数据元素的关键字值与给定值 key 进行比较,若某个数据元素的关键字值和给定值相等,则查找成功,否则查找失败。顺序查找又叫线性查找。

【算法 8.1】 顺序查找。

```
def seqSearch(self,key):
    for i in range(self.len):
        if self.list[i].key == key:
            return i # 返回关键字值与给定值相等的数据元素的标
    return - 1
```

2. 算法性能分析

假设查找每个数据元素的概率相等,对于一个长度为 n 的顺序表,其平均查找长度如下:

$$\mathrm{ASL} = \sum_{i=0}^{n-1} p_i c_i = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{n+1}{2}$$

若查找失败,关键字比较次数为 n,因此顺序查找的时间复杂度为 $O(n)$。

8.2.2 二分查找

1. 二分查找算法的实现

二分查找是对有序表进行的查找。通常假定有序表按关键字值从小到大有序排列,二分查找首先取整个表的中间数据元素的关键字值和给定值 key 进行比较,若相等,则查找成功;若给定值小于该元素的关键字值,则在左子表中重复上述步骤;若给定值大于该元素的关键字值,则在右子表中重复上述步骤,直到找到关键字值为 key 的记录或子表长度为 0。二分查找又叫折半查找。

假设有序表的数据元素的关键字序列为{2,7,13,23,45,67,89,90,92},当给定的 key 值为 23 时进行二分查找的过程如图 8.1 所示。

```
[2  7  13  23  45  67  89  90  92]
               ↑mid
[2  7  13  23  45]  67  89  90  92
       ↑mid
 2  7 [13  23  45]  67  89  90  92
           ↑mid
```

图 8.1 二分查找的过程

【算法 8.2】 二分查找。

```
def binarySearch(self,key):
```

```
    if self.len > 0:
        # 查找表的上界与下界
        low = 0
        high = self.len - 1
        while low <= high:
            mid = (low + high)//2 # 取中间元素位置
            if self.list[mid].key == key:
                return mid
            elif self.list[mid].key < key: # 查找范围为后半部分
                low = mid + 1
            else: # 查找范围为前半部分
                high = mid - 1
    return -1
```

2. 算法性能分析

假设查找每个数据元素的概率相等,对于一个长度为 $n=2k-1$ 的有序表,线性表最多被平分 $k=\log_2(n+1)$次即可完成查找。又因为在 i 次查找中可以找到的元素个数为 $2i-1$ 个,所以其平均查找长度如下:

$$\text{ASL} = \sum_{i=0}^{k} p_i c_i = \frac{1}{n}\sum_{i=0}^{k}(i \times 2^{i-1}) = \log_2(n+1)-1+\frac{1}{n}\log_2(n+1) \approx \log_2(n+1)-1$$

因此,查找的时间复杂度为 $O(\log_2 n)$。

8.2.3 分块查找

分块查找是将线性表分为若干块,块之间是有序的,块中的元素不一定有序,将每块中最大的关键字值按块的顺序建立索引顺序表,在查找时首先通过索引顺序表确定待查找元素可能所在的块,然后在块中寻找该元素。

索引顺序表是有序表,可以采用顺序查找或者二分查找;块中元素无序,必须采用顺序查找。

假设线性表中数据元素的关键字为{23,12,3,4,5,56,75,24,44,33,77,76,78,90,98},有 15 个节点,被分为 3 块,则要求每一块中的最大关键字值小于后一块中的最小关键字值。分块有序表的索引存储表如图 8.2 所示。

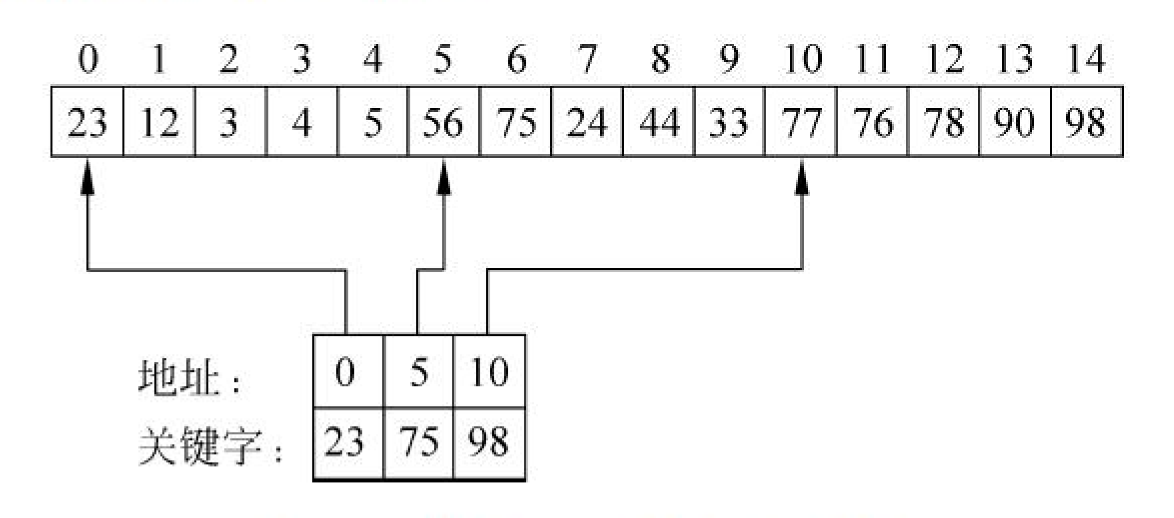


图 8.2 分块有序表的索引存储表

由于分块查找是顺序查找和二分查找的结合,因此分块查找的平均查找长度为查找索引表确定元素所在块的平均查找长度 L_b 加上在块中查找元素的平均查找长度 L_c,可表示如下:

$$\text{ASL} = L_b + L_c$$

一般将长度为 n 的线性表均匀分成 b 块，每块中含有 s 个元素，即 $b=\left\lceil \frac{n}{s} \right\rceil$。假定查找每个元素的概率相同，若使用顺序查找确定元素所在的块，则分块查找的平均查找长度如下：

$$\text{ASL} = L_b + L_c = \frac{1}{b}\sum_{i=0}^{b-1}(i+1) + \frac{1}{s}\sum_{i=0}^{s-1}(i+1) = \frac{1}{2}\left(\frac{n}{s}+s\right)+1$$

若使用二分查找确定元素所在的块，则分块查找的平均查找长度如下：

$$\text{ASL} \approx \log_2\left(\frac{n}{s}+1\right)+\frac{s}{2}$$

8.3 动态表查找

动态查找表是指对表的操作包括对表的修改的表，即表结构本身实际是在查找过程中动态生成的。动态查找表有多种不同的实现方法，本节中只讨论在各种树结构上查找的实现方法。

8.3.1 二叉排序树查找

1. 二叉排序树的概念

二叉排序树是具有下列性质的二叉树。

(1) 若右子树非空，则右子树上所有节点的值均大于根节点的值。

(2) 若左子树非空，则左子树上所有节点的值均小于根节点的值。

(3) 左、右子树也为二叉排序树。

二叉排序树可以为空树，其结构如图 8.3 所示。

【例 8.1】 一棵二叉排序树的结构如图 8.4(a)所示，节点的值为 1～8，请标出各节点的值。

解：

由二叉排序树的概念可得二叉排序树中各节点的值如图 8.4(b)所示。

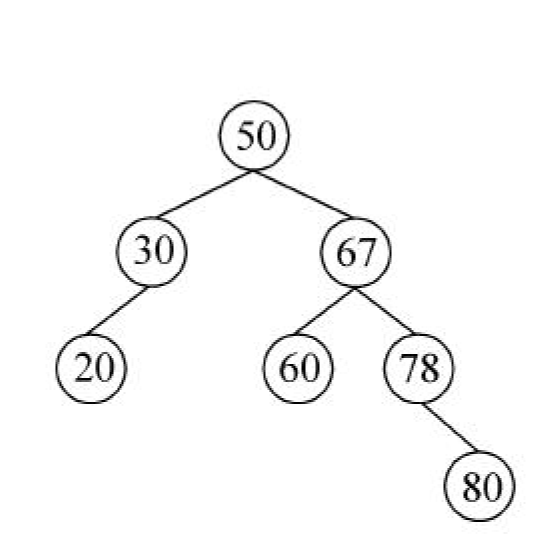
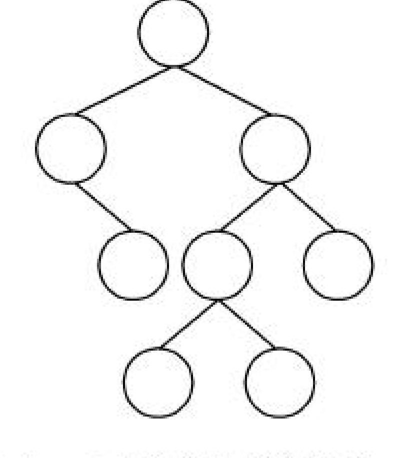
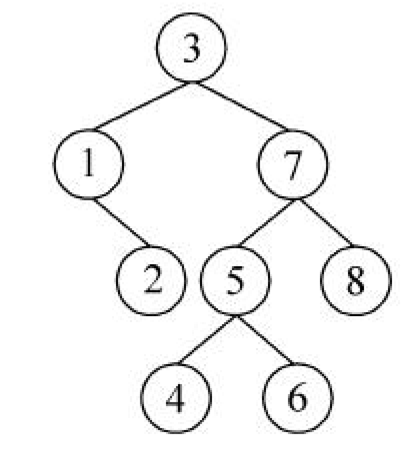


图 8.3 二叉排序树



(a) 二叉排序树的结构



(b) 各节点的值

图 8.4 二叉排序树的结构以及各节点的值

2. 二叉排序树查找算法的实现

二叉排序树查找过程的主要步骤如下。

(1) 若查找树为空，则查找失败。

(2) 若查找树非空,且给定值 key 等于根节点的关键字值,查找成功。

(3) 若查找树非空,且给定值 key 小于根节点的关键字值,在根节点的左子树上进行查找过程。

(4) 若查找树非空,且给定值 key 大于根节点的关键字值,在根节点的右子树上进行查找过程。

以二叉链表作为二叉排序树的存储结构,其节点类定义如下:

```
class BiTreeNode(object):
    def __init__(self,key,data,lchild=None,rchild=None):
        self.key = key            # 节点关键字值
        self.data = data          # 节点的数据值
        self.lchild = lchild      # 节点的左孩子
        self.rchild = rchild      # 节点的右孩子
```

二叉排序树的类结构定义如下:

```
class BSTree(object):
    def __init__(self,root=None):
        self.root = root # 树的根节点
```

【算法 8.3】 二叉排序树查找。

```
def search(self,key):
    return self.searchBST(key,self.root)

def searchBST(self,key,p):
    if p is None: # 查找树为空,查找失败
        return None
    if key==p.key: # 查找成功
        return p.data
    elif key<p.key: # 在左子树中查找
        return self.searchBST(key,p.lchild)
    else: # 在右子树中查找
        return self.searchBST(key,p.rchild)
```

3. 二叉排序树插入算法的实现

在向二叉排序树中插入一个节点时首先对二叉排序树进行查找,若查找成功,则节点已存在,不需要插入;若查找失败,再将新节点作为叶子节点插入到二叉排序树中。

构造二叉排序树是从空树开始逐个插入节点的过程。假设关键字序列为{23,56,73,34,12,67},则构造二叉排序树的过程如图 8.5 所示。

【算法 8.4】 二叉排序树插入算法。

```
def insert(self,key,data):
    p = BiTreeNode(key,data) # 为元素建立节点
    if self.root is None: # 若根节点为空,建立新的根节点
        self.root = p
    else:
        self.insertBST(self.root,p)

```

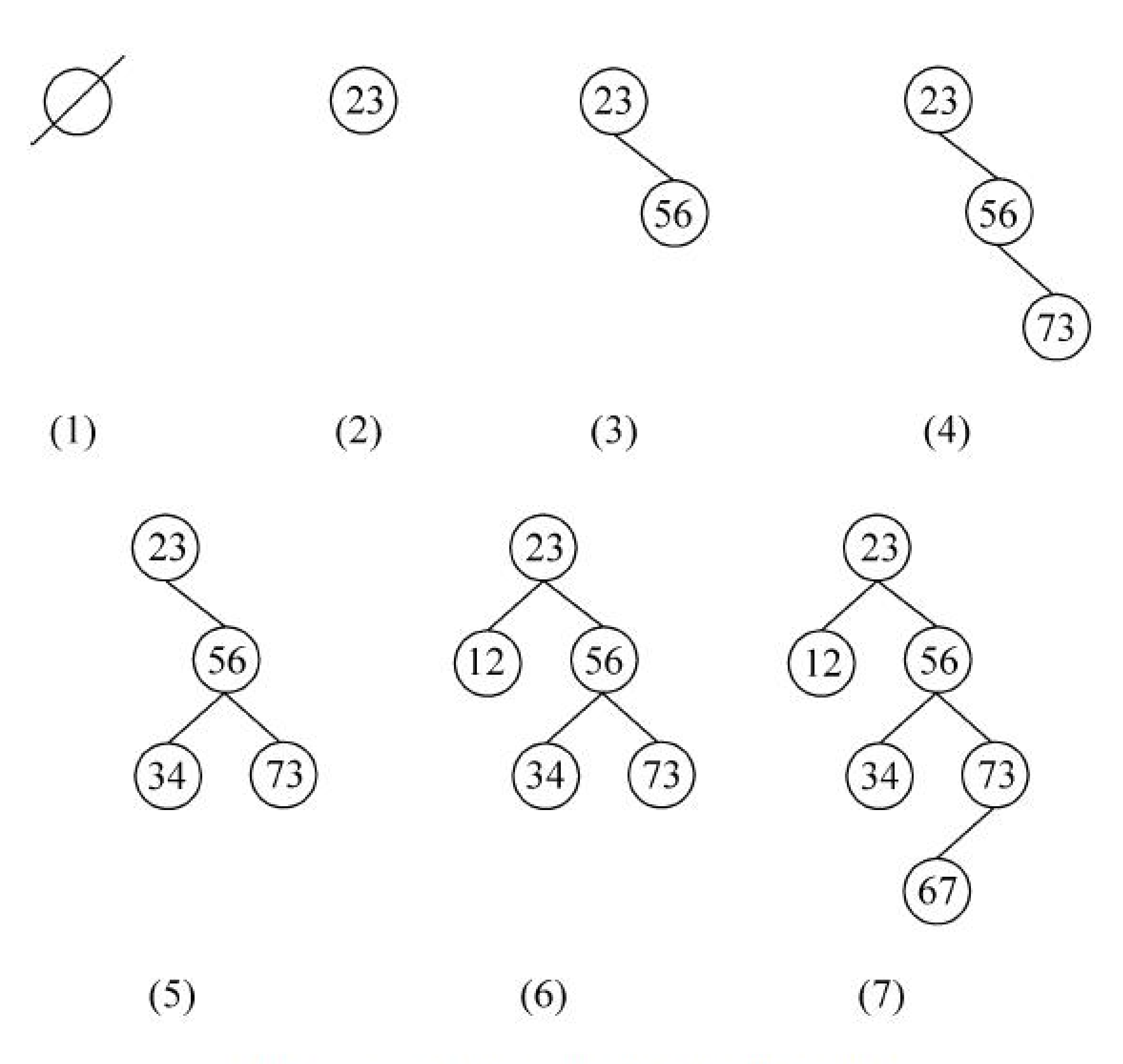


图 8.5　构造二叉排序树的过程

```
def insertBST(self,r,p):
    if r.key<p.key: # 查找右子树
        if r.rchild is None:
            r.rchild = p
        else:
            self.insertBST(r.rchild,p)
    else: # 查找左子树
        if r.lchild is None:
            r.lchild = p
        else:
            self.insertBST(r.lchild,p)
```

4. 二叉排序树删除算法的实现

在二叉排序树中删除一个元素要保证删除后的树仍然是二叉排序树，分为 3 种情况进行讨论。

(1) 若待删除的节点是叶子节点，可直接删除。

(2) 若待删除的节点只有左子树或右子树，则将左子树或右子树的根节点代替被删除节点的位置。

(3) 若待删除的节点有左、右两棵子树，在中序遍历下则将待删除节点的前驱节点或后继节点代替被删除节点的位置，并将该节点删除。

【算法 8.5】 二叉排序树删除算法。

```
def remove(self,key):
    # 删除关键字为 key 的节点
    self.removeBST(key,self.root,None)

def removeBST(self,key,p,parent):
    if p is None: # 树空，直接返回
        return
```

```
        if p.key > key: # 在左子树中删除
            self.removeBST(key,p.lchild,p)
        elif p.key < key: # 在右子树中删除
            self.removeBST(key,p.rchild,p)
        elif p.lchild is not None and p.rchild is not None: # 删除此节点,左右子树非空
            inNext = p.rchild
            while inNext.lchild is not None:
                inNext = inNext.lchild
            p.data = inNext.data
            p.key = inNext.key
            self.removeBST(p.key,p.rchild,p)
        else: # 只有一棵子树或者没有子树
            if parent is None:
                if p.lchild is not None:
                    self.root = p.lchild
                else:
                    self.root = p.rchild
                return
            if p == parent.lchild:
                if p.lchild is not None:
                    parent.lchild = p.lchild
                else:
                    parent.lchild = p.rchild
            elif p == parent.rchild:
                if p.lchild is not None:
                    parent.rchild = p.lchild
                else:
                    parent.rchild = p.rchild
```

二叉排序树删除操作的时间主要花费在查找待删除节点和查找被删除节点的后继节点上,查找操作与二叉排序树的深度有关,对于按给定序列建立的二叉排序树,若其左、右子树均匀分布,查找过程类似于有序表的二分查找,时间复杂度为 $O(\log_2 n)$;但若给定序列原来有序,则建立的二叉排序树为单链表,其查找效率和顺序查找一样,时间复杂度为 $O(n)$。

【例 8.2】 将数列{24,15,38,27,121,76,130}的各元素依次插入一棵初始为空的二叉排序树中,请画出最后的结果并求等概率情况下查找成功的平均查找长度。

解:

二叉排序树如图 8.6 所示,其平均查找长度=1+2×2+3×2+4×2=19/7。

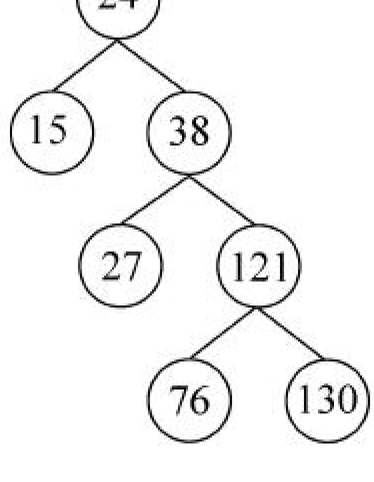


图 8.6 二叉排序树

8.3.2 平衡二叉树

1. 平衡二叉树的概念

上一节中讨论了对二叉排序树进行查找操作的时间复杂度,若二叉排序树的左、右子树均匀分布,查找操作的时间复杂度为 $O(\log_2 n)$;若给定序列原来有序,二叉排序树为单链

表,查找操作的时间复杂度为 $O(n)$。所以,为了提高二叉排序树的查找效率,在构造二叉排序树的过程中若出现左、右子树分布不均匀的现象,我们将对其进行调整,使其保持均匀,即此时的二叉排序树为平衡二叉树。

平衡二叉树是左、右子树深度之差的绝对值小于 2 并且左、右子树均为平衡二叉树的树。平衡二叉树又叫 AVL 树,可以为空,其某个节点的左子树深度与右子树深度之差称为该节点的平衡因子或平衡度。

2. 平衡二叉树的实现

在平衡二叉树上删除或插入节点后可能会使二叉树失去平衡。对非平衡二叉树的调整可依据失去平衡的原因分为以下 4 种情况进行(假设在平衡二叉树上因插入新节点而失去平衡的最小子树的根节点为 A)。

1) LL 型平衡旋转(单向右旋)

原因:在 A 的左孩子的左子树上插入新节点,使 A 的平衡度由 1 变为 2,以 A 为根的子树失去平衡。

调整:提升 A 的左孩子 B 为新子树的根节点,A 为 B 的右孩子,同时将 B 的右子树 BR 调整为 A 的左子树,如图 8.7 所示。

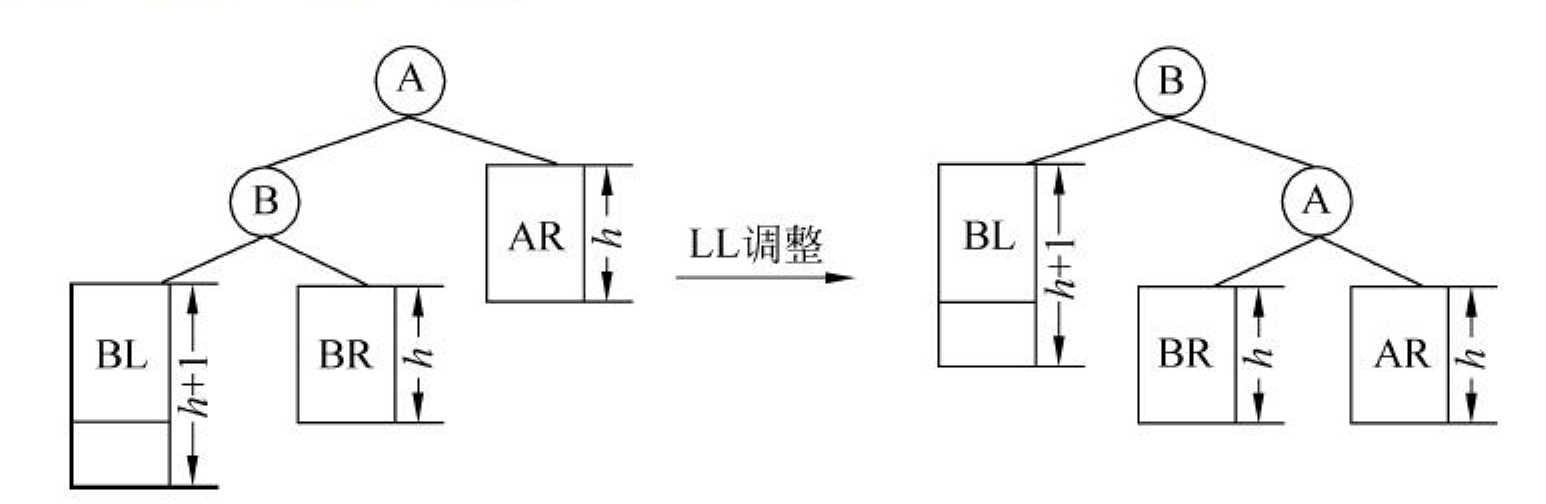


图 8.7 LL 型平衡旋转

2) RR 型平衡旋转(单向左旋)

原因:在 A 的右孩子的右子树上插入新节点,使 A 的平衡度由 −1 变为 −2,以 A 为根的子树失去平衡。

调整:提升 A 的右孩子 B 为新子树的根节点,A 为 B 的左孩子,同时将 B 的左子树 BL 调整为 A 的右子树,如图 8.8 所示。

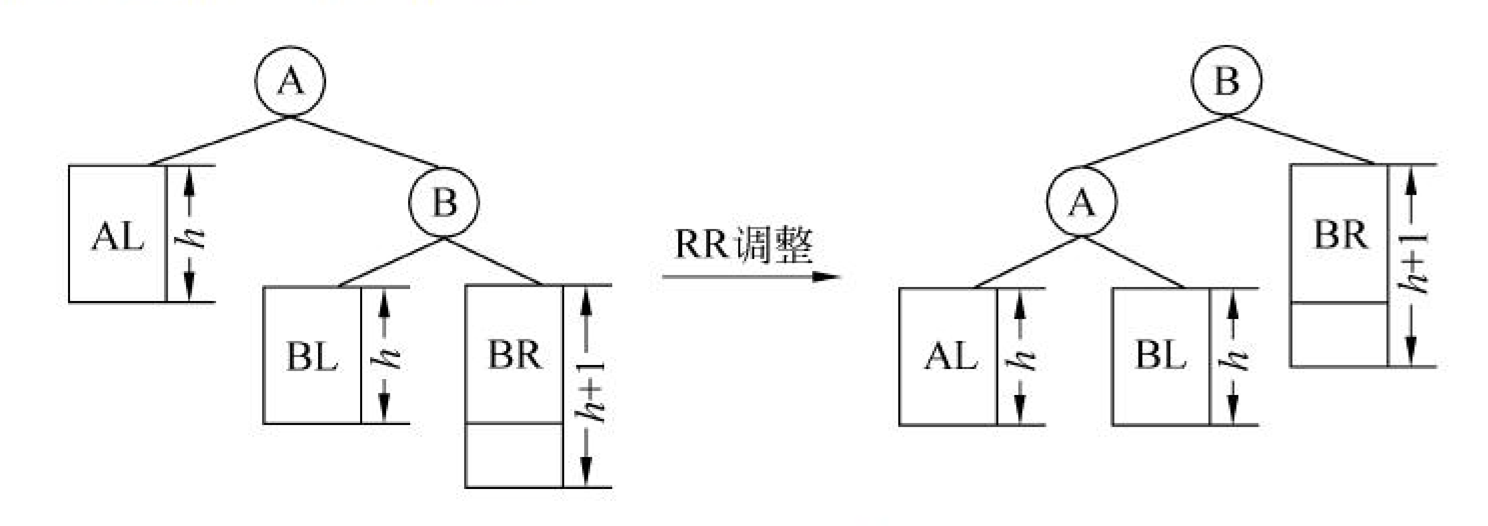


图 8.8 RR 型平衡旋转

3) LR 型平衡旋转(先左旋后右旋)

原因:在 C 的左孩子的右子树上插入新节点,使 C 的平衡度由 1 变为 2,以 C 为根的子树失去平衡。

调整：提升 C 的左孩子 A 的右孩子 B 为新子树的根节点，C 为 B 的右孩子，A 为 B 的左孩子，将 B 的左子树 BL 调整为 A 的右子树，将 B 的右子树 BR 调整为 C 的左子树，如图 8.9 所示。

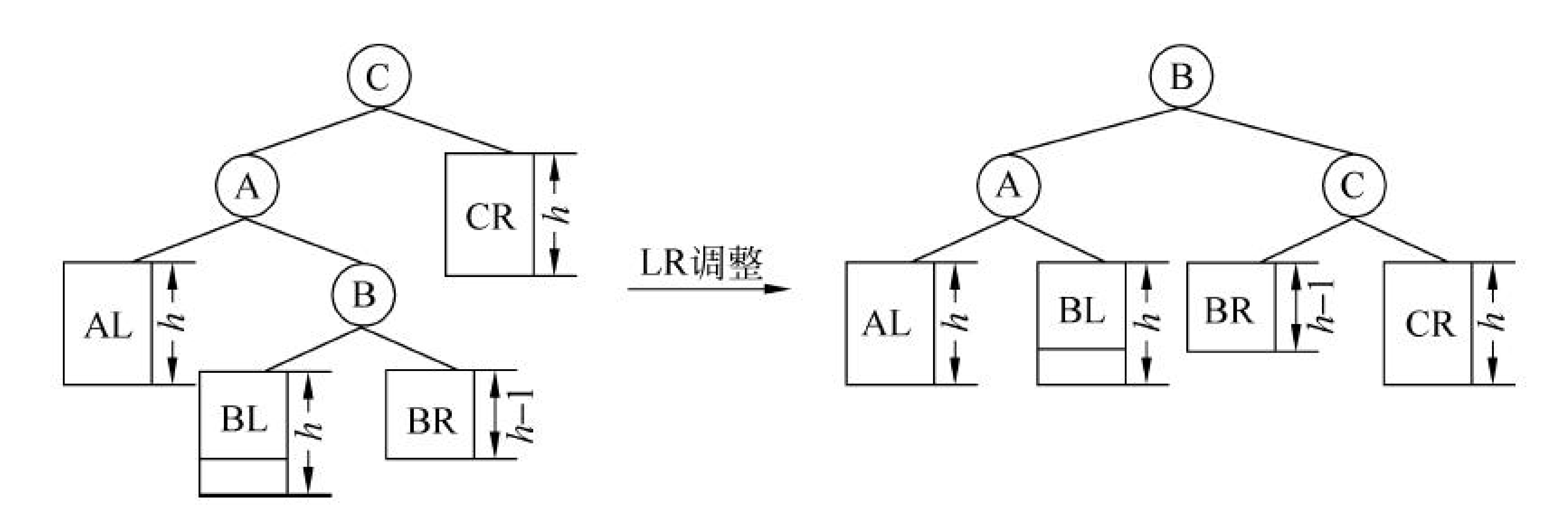


图 8.9　LR 型平衡旋转

4) RL 型平衡旋转(先右旋后左旋)

原因：在 A 的右孩子的左子树上插入新节点，使 A 的平衡度由 −1 变为 −2，以 A 为根的子树失去平衡。

调整：提升 A 的右孩子 C 的左孩子 B 为新子树的根节点，A 为 B 的左孩子，C 为 B 的右孩子，将 B 的左子树 CL 调整为 A 的右子树，将 B 的右子树 BR 调整为 C 的左子树，如图 8.10 所示。

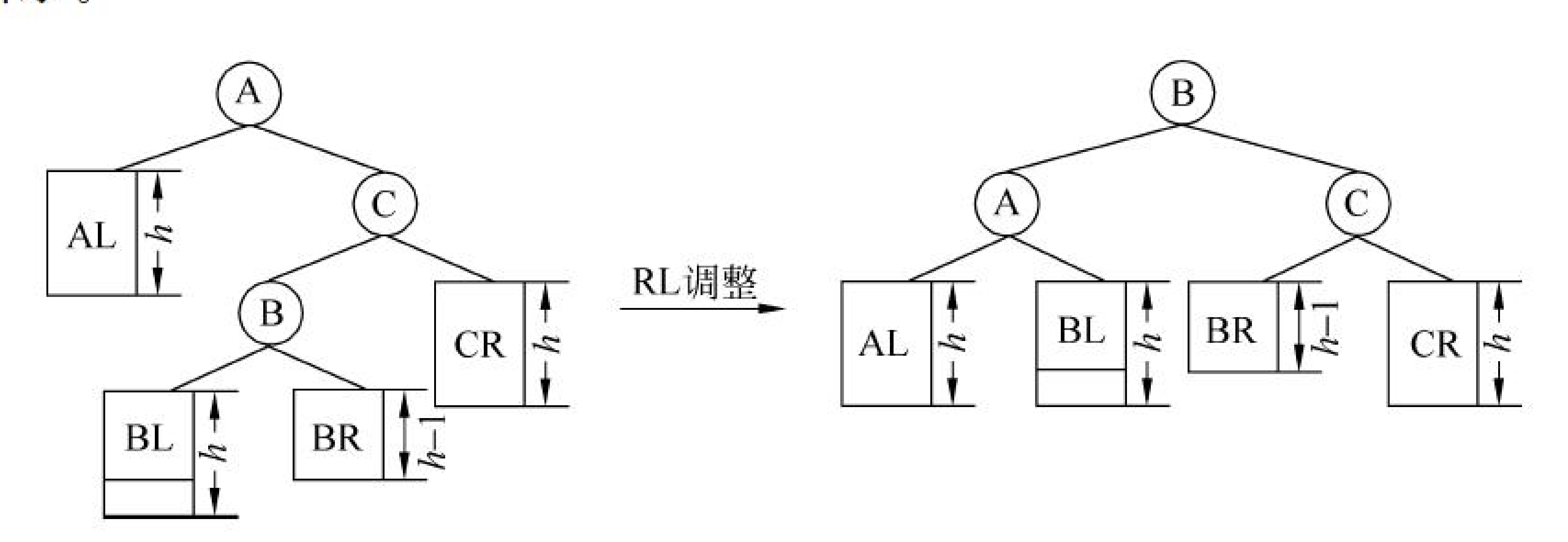


图 8.10　RL 型平衡旋转

采用平衡二叉树提高了查找操作的速度，但是使插入和删除操作复杂化，因此平衡二叉树适用于二叉排序树一经建立就很少进行插入和删除操作而主要进行查找操作的场合中，其查找的时间复杂度为 $O(\log_2 n)$。

【例 8.3】 试推导含有 12 个节点的平衡二叉树的最大深度，并画出一棵这样的树。

解：

令 F_k 表示含有最少节点的深度为 k 的平衡二叉树的节点树目，则 $F_1=1, F_2=2, \cdots, F_n=F_{n-2}+F_{n-1}+1$。含有 12 个节点的平衡二叉树的最大深度为 5，如图 8.11 所示。

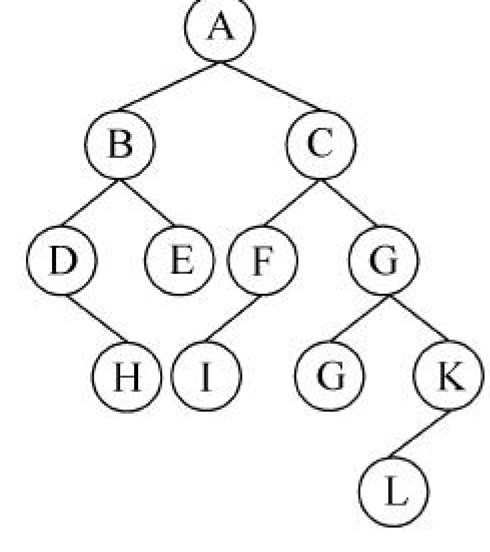


图 8.11　平衡二叉树

8.3.3　B⁻树和 B⁺树

1. B⁻树的概念

B⁻树是一种平衡的多路查找树，在文件系统中 B⁻树已经成为索引文件的一种有效结

构。一棵 m 阶的 B^- 树是满足下列特征的 m 叉树。

(1) 树中的每个节点最多有 m 棵子树。

(2) 若根节点不是叶子节点,则至少有两棵子树。

(3) 所有的非终端节点包含信息($n,P_0,K_1,P_1,K_2,P_2,\cdots,K_n,P_n$)。其中,$K_i(1\leqslant i\leqslant n)$为关键字,且 $K_i<K_{i+1}$;$P_j(0\leqslant j<n)$是指向子树根节点的指针且 P_j所指子树中所有节点的关键字值都小于 K_{j+1},P_n所指子树中所有节点的关键字值均大于 K_n。

2. B^+树的概念

B^+ 和 B^- 树的结构大致相同,一颗 m 阶的 B^- 树和一颗 m 阶的 B^+ 树的差异在于:

(1) 在 B^- 树中,每一个节点含有 n 个关键字和 $n+1$ 棵子树;而在 B^+ 树中,每一个节点含有 n 个关键字和 n 棵子树。

(2) 在 B^- 树中,每个节点中的关键字个数 n 的取值范围是 $m/2-1\leqslant n\leqslant m-1$;而在 B^+ 树中,每个节点中的关键字个数 n 的取值范围是 $m/2\leqslant n\leqslant m$,树的根节点的关键字个数的取值范围是 $1\leqslant n\leqslant m$。

(3) B^+ 树中的所有叶子节点包含了全部关键字及指向对应记录的指针,且所有叶子节点按关键字值从小到大的顺序依次链接。

(4) B^+ 树中所有非叶子节点仅起到索引的作用,即节点中的每一个索引项只含有对应子树的最大关键字和指向该子树的指针,不含有该关键字对应记录的存储地址。

8.4 哈希表查找

8.4.1 哈希表的概念

哈希存储是以关键字值为自变量通过一定的函数关系(称为散列函数或者哈希函数)计算出数据元素的存储地址,并将该数据元素存入到相应地址的存储单元。在查找时只需要根据查找的关键字采用同样的函数计算出存储地址即可到相应的存储单元取得数据元素。

对于含有 n 个数据元素的集合,总能找到关键字与哈希地址一一对应的函数。若选取函数 $f(\mathrm{key})=\mathrm{key}$,数据元素中的最大关键字为 m,需要分配 m 个存储单元,由于关键字集合比存储空间大得多,可能造成存储空间的很大浪费。此外,通过哈希函数变换后可能将不同的关键字映射到同一个哈希地址上,这种现象叫冲突。所以使用哈希方法进行查找时需要关注两个问题,一是要构造好的哈希函数,尽量加快地址计算速度,减少存储空间的浪费;二是制定解决冲突的方法。

根据哈希函数和处理冲突的方法,将一组关键字映射到一个有限的、地址连续的地址集合空间上,并且数据元素的存储位置由关键字通过哈希函数计算得来,这样的表称为哈希表。

8.4.2 哈希函数

哈希函数的构造需要遵循以下两个原则。

(1) 尽可能将关键字均匀地映射到地址集合空间,减少存储空间的浪费。

(2) 尽可能降低冲突发生的概率。

下面介绍 6 种常用的哈希函数。

1. 直接地址法

直接地址法即：

$$H(\text{key})=a\times \text{key}+b$$

它是取关键字的某个线性函数值为哈希地址。

直接地址法简单，不会产生冲突，但是关键字值往往是离散的，且关键字集合比哈希地址大，会造成存储空间的浪费。

2. 除留余数法

除留余数法即：

$$H(\text{key})=\text{key}\%p \quad (p\leqslant m)$$

它是以关键字除 p 的余数作为哈希地址，其中 m 为哈希表长度。

使用除留余数法，p 的选择很重要，否则会造成严重冲突。例如，若取 $p=2^k$，则 $H(\text{key})=\text{key}\%p$ 的值仅仅是用二进制表示的 key 右边的 k 个位，造成了关键字的映射并不均匀，易造成冲突。通常，为了获得比较均匀的地址分布，一般令 p 为小于或等于 m 的某个最大素数。

3. 数字分析法

数字分析法是对关键字的各位进行分析，丢掉分布不均匀的位，留下分布均匀的位作为哈希地址。对于不同的关键字集合，所保留的地址可能不相同，因此这种方法主要应用于关键字的位数比存储区域的地址码位数多的情况，并且在使用时需要能预先估计出全体关键字的每一位上各种数字出现的频度的情况。

4. 平方取中法

平方取中法是取关键字平方的中间几位作为哈希地址的方法。一个数的平方值的中间几位和数的每一位都有关系，因此平方取中法得到的哈希地址和关键字的每一位都有关系，使得哈希地址的分布较为均匀。

平方取中法适用于关键字中的每一位取值都不够分散或者较分散的位数小于哈希地址所需要的位数的情况。

5. 折叠法

折叠法是将关键字自左向右或自右向左分成位数相同的几部分，最后一部分位数可以不同，然后将这几部分叠加求和，并按哈希表的表长取最后几位作为哈希地址。常用的折叠法有以下两种。

(1) 移位叠加法：将分割后的各部分的最低位对齐，然后相加。

(2) 间界叠加法：从一端向另一端沿分割界来回折叠后对齐最后一位相加。

折叠法适用于位数较多，并且每一位的取值都分散均匀的情况。

6. 随机数法

随机数法是取关键字的随机数函数值为它的哈希地址，即 $H(\text{key})=\text{random}(\text{key})$。此方法主要适用于关键字长度不相等的情况。

8.4.3 解决冲突的方法

选取好的哈希函数可以减少冲突发生的概率，但是冲突是不可避免的。本节介绍 4 种常用的解决哈希冲突的方法。

1. 开放定址法

开放定址法是当冲突发生时形成一个地址序列，沿着这个地址序列逐个探测，直到找到一个空的开放地址，将发生冲突的数据存放到该地址中。

地址序列的值可表示如下：

$$H_i=(H(\text{key})+d_i)\%m \quad (i=1,2,\cdots,k;\ k\leqslant m-1)$$

其中 $H(\text{key})$是关键字值为 key 的哈希函数，m 为哈希表长，d_i 为每次探测时的地址增量。

根据地址增量取值的不同可以得到不同的开放地址处理冲突探测方法，主要分为以下 3 种。

1）线性探测法

线性探测法的地址增量如下：

$$d_i = 1,2,\cdots,m-1$$

其中 i 为探测次数。这种方法在解决冲突时，依次探测下一个地址，直到找到一个空的地址，若在整个空间中都找不到空地址将产生溢出。

利用线性探测法解决冲突问题容易造成数据元素的“聚集”，即多个哈希地址不同的关键字争夺同一个后继哈希地址。假设表中的第 $i,i+1,i+2$ 地址非空，则下一次哈希地址为 $i,i+1,i+2$ 的数据都企图填入到 $i+3$ 的位置处。这种现象发生的根本原因是查找序列过分集中在发生冲突的存储单元后面，没有在整个哈希表空间分散开来。

2）二次探测法

二次探测法的地址增量如下：

$$d_i = 1^2,-1^2,2^2,-2^2,\cdots,k^2,-k^2 \quad \left(k\leqslant \frac{m}{2}\right)$$

其中 m 为哈希表长。这种方法能够避免“聚集”现象的发生，但是不能探测到哈希表上的所有存储单元。

3）双哈希函数探测法

双哈希函数探测法是使用另外一个哈希函数 RH(key)计算地址增量。哈希地址的计算方法可以表示如下：

$$H_i = (H(\text{key})+i\times \text{RH}(\text{key}))\%m \quad (i = 1,2,\cdots,m-1)$$

这种方法也可以避免“聚集”现象的发生。

2. 链地址法

链地址法是将所有具有相同哈希地址的不同关键字的数据元素链接到同一个单链表中。若哈希表的长度为 m，则可将哈希表定义为一个由 m 个头指针组成的指针数组 $T[0..m-1]$，凡是哈希地址为 i 的数据元素均以节点的形式插入以 $T[i]$为头指针的链表中。

假设一组数据元素的关键字序列为{2,4,6,7,9}，按照哈希函数 $H(\text{key})=\text{key}\%4$ 和链地址法处理冲突得到的哈希表如图 8.12 所示。

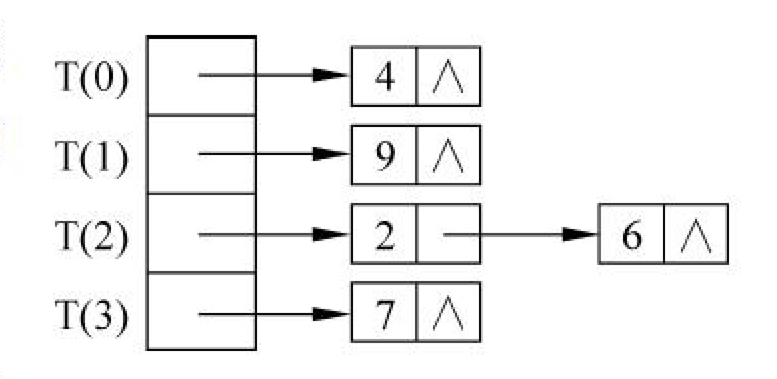


图 8.12 用链地址法处理冲突所得的哈希表

3. 公共溢出区法

公共溢出区法是另建一个溢出表，当不发生冲突时数据元素存入基本表，当发生冲突时数据元素存入溢出表。

4. 再哈希法

再哈希法是当发生冲突时再使用另一个哈希函数得到一个新的哈希地址,若再发生冲突,则再使用另一个函数,直到不发生冲突为止。此种方法需要预先设计一个哈希函数序列:

$$H_i = \mathrm{RH}_i(\mathrm{key}) \quad (i = 1,2,\cdots,k)$$

这种方法不易产生“聚集”现象,但会增加计算的时间。

8.4.4 哈希表查找性能分析

在哈希表上进行查找的过程和建立哈希表的过程一致,并且插入和删除操作的时间也取决于查找进行的时间,因此本节中只分析哈希表查找操作的性能。

使用平均查找长度来衡量哈希表的查找效率,在查找过程中与关键字的比较次数取决于哈希函数的选取和处理冲突的方法。假设哈希函数是均匀的,即对同样一组随机的关键字出现冲突的可能性是相同的。因此,哈希表的查找效率主要取决于处理冲突的方法。发生冲突的次数和哈希表的装填因子有关,哈希表的装填因子如下:

$$\alpha = \frac{\text{哈希表中的数据元素个数}}{\text{哈希表的长度}}$$

填入表中的数据元素越多,α 越大,产生冲突的可能性越大;填入表中的数据元素越少,α 越小,产生冲突的可能性越小。α 通常取 1 和 1/2 之间的较小的数。

表 8.1 中给出了不同处理冲突的方法的平均查找长度。

表 8.1 不同的处理冲突方法的平均查找长度

处理冲突的方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$U_{nl} \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$
二次探测法	$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr} \approx \frac{1}{1-\alpha}$
双哈希法	$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr} \approx \frac{1}{1-\alpha}$
链地址法	$S_{nc} \approx 1+\frac{\alpha}{2}$	$U_{nc} \approx \alpha + e^{-\alpha}$

由表 8.1 可见,哈希表的平均查找长度是 α 的函数,因此总可以选择一个合适的装填因子 α,可将平均查找长度限定在一个范围内。

【例 8.4】 已知哈希函数 $H(k)=k \bmod 12$,键值序列为{25,37,52,43,84,99,120,15,26,11,70,82},采用链地址法处理冲突,试构造哈希表,并计算查找成功的平均查找长度。

解：

$H(25)=1, H(37)=1, H(52)=4, H(43)=7, H(84)=0, H(99)=3,$

$H(120)=0, H(15)=3, H(26)=2, H(11)=11, H(70)=10, H(82)=10$

构造的散列表如图 8.13 所示。

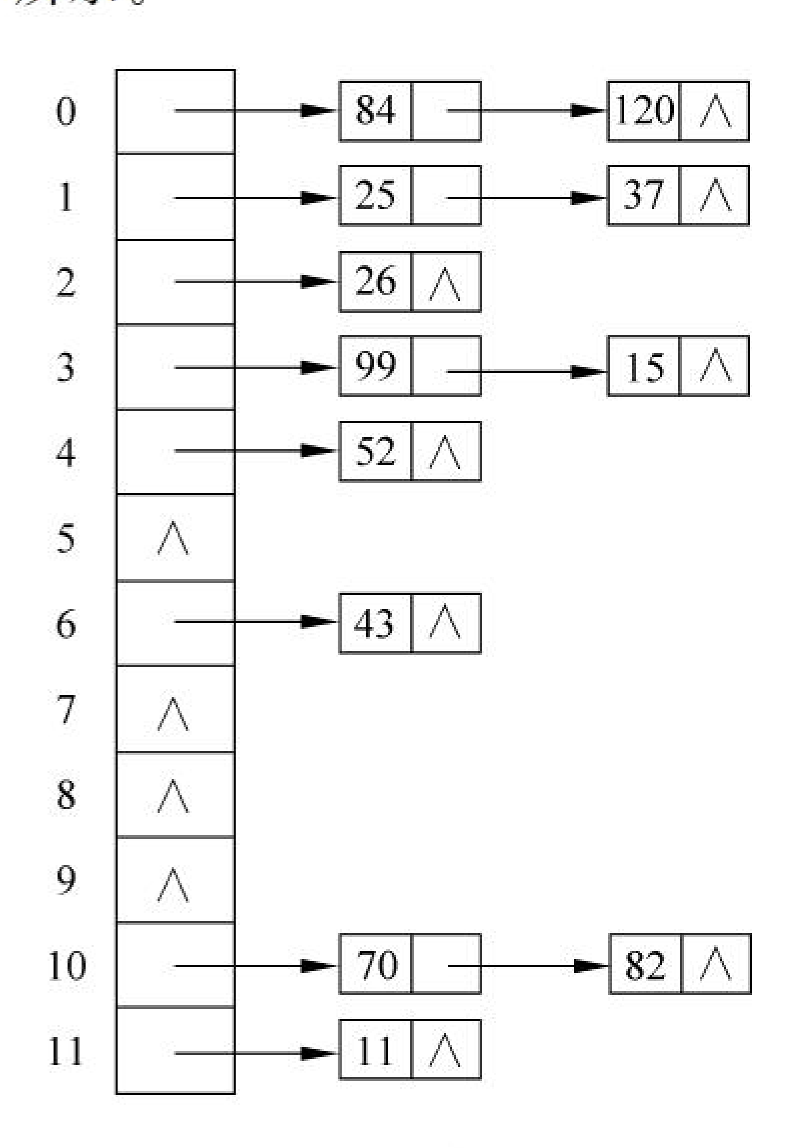


图 8.13　散列表

平均查找长度 ASL=(8×1+4×2)/12=16/12。

【例 8.5】 已知关键码序列为{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}，散列表的地址空间为 0～16，设散列函数为 $H(x)=i/2$（向下取整），其中 i 为关键码中第一个字母在字母表中的序号，采用线性探测法和链地址法处理冲突，试分别构造散列表，并求等概率情况下查找成功的平均查找长度。

解：

H(Jan)=10/2=5，H(Feb)=6/2=3，H(Mar)=13/2=6，

H(Apr)=1/2=0，H(May)=13/2=6，H(Jun)=10/25，

H(Jul)=10/25，H(Aug)=1/2=0，H(Sep)=19/2=8，

H(Oct)=15/2=7，H(Nov)=14/2=7，H(Dec)=4/2=2

采用线性探测法处理冲突得到的闭散列表如图 8.14 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Apr	Aug	Dec	Feb		Jan	Mar	May	Jun	Jul	Sep	Oct	Nov				

图 8.14　采用线性探测法处理冲突得到的闭散列表

平均查找长度=(1+1+1+1+2+4+5+2+3+5+6+1)/12=32/12。

采用链地址法处理冲突得到的开散列表如图 8.15 所示。

平均查找长度=(1×7+2×4+3×1)/12=18/12。

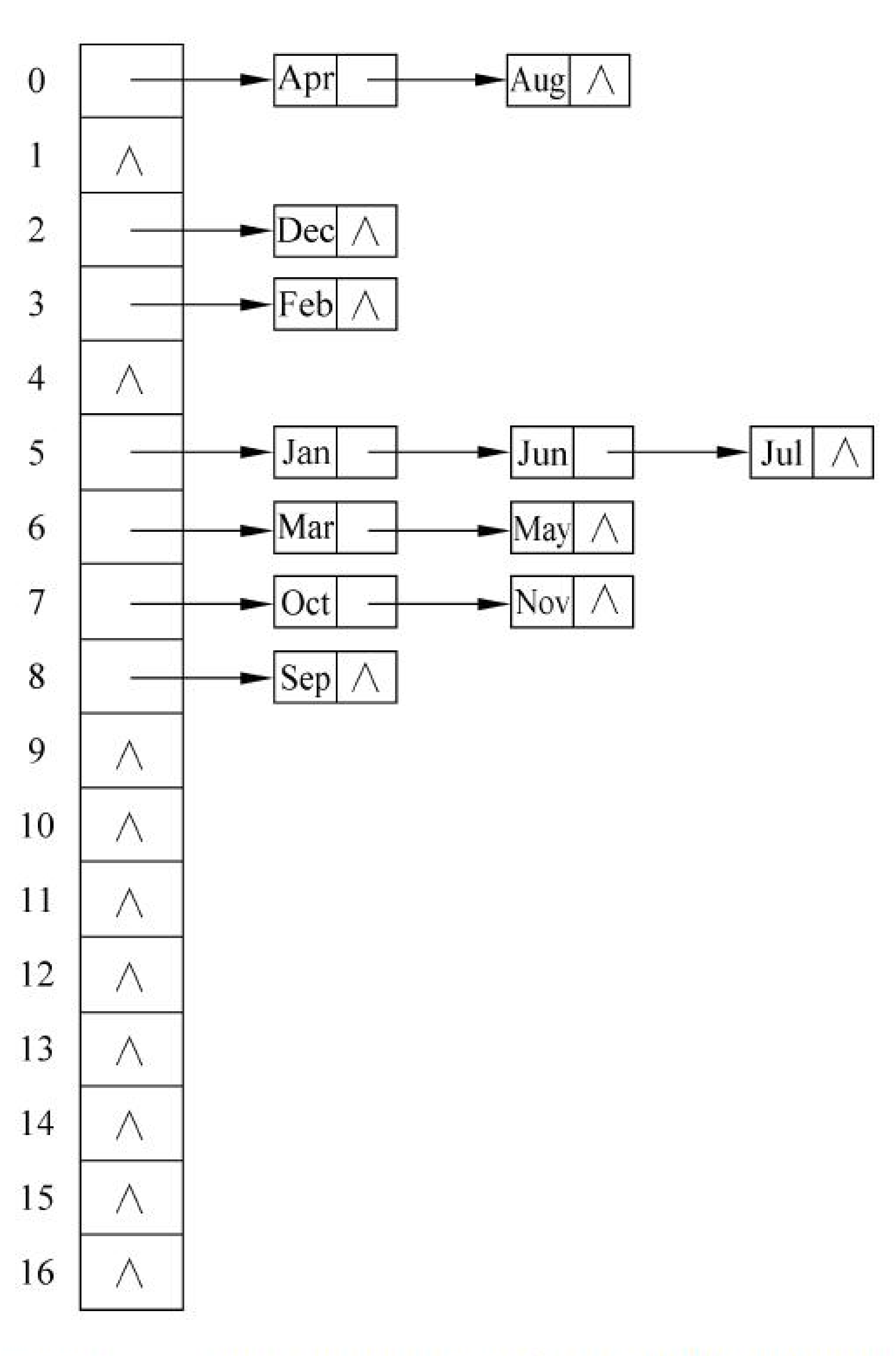


图 8.15 采用链地址法处理冲突得到的开散列表

小 结

(1) 查找就是在由一组记录组成的集合中寻找属性值符合特定条件的数据元素。若集合中存在符合条件的记录,则查找成功,否则查找失败。

(2) 查找表是一种以同一类型的记录构成的集合为逻辑结构、以查找为核心运算的灵活的数据结构。在实现查找表时要根据实际情况按照查找的具体要求组织查找表,从而实现高效率的查找。

(3) 静态查找表是指对表的操作不包括对表的修改的表,可以用顺序表或线性链表进行表示,分为顺序查找、二分查找和分块查找 3 种。

(4) 动态查找表是指对表的操作包括对表的修改的表,即表结构本身是在查找过程中动态生成的。动态查找表有多种不同的实现方法,如二叉排序树查找。

(5) 平衡二叉树是左、右子树深度之差的绝对值小于 2 并且左、右子树均为平衡二叉树的树。平衡二叉树又叫 AVL 树。

(6) 哈希存储以关键字值为自变量,通过哈希函数计算出数据元素的存储地址,并将该数据元素存入到相应地址的存储单元。在进行哈希表查找时只需要根据查找的关键字采用同样的函数计算出存储地址即可到相应的存储单元取得数据元素。在进行哈希表查找时需要构造好的哈希函数并且制定解决冲突的方法。

习 题 8

一、选择题

1. 已知一个有序表为{12,18,24,35,47,50,62,83,90,115,134},当折半查找值为 90 的元素时经过(　　)次比较后查找成功。

A. 2　　B. 3　　C. 4　　D. 5

2. 已知 10 个元素{54、28、16、73、62、95、60、26、43},按照依次插入的方法生成一棵二叉排序树,查找值为 62 的节点所需的比较次数为(　　)。

A. 2　　B. 3　　C. 4　　D. 5

3. 已知数据元素{34、76、45、18、26、54、92、65},按照依次插入节点的方法生成一棵二叉排序树,则该树的深度为(　　)。

A. 4　　B. 5　　C. 6　　D. 7

4. 按(　　)遍历二叉排序树得到的序列是一个有序序列。

A. 前序　　B. 中序　　C. 后序　　D. 层次

5. 一棵高度为 h 的平衡二叉树最少含有(　　)个节点。

A. $2h$　　B. $2h-1$　　C. $2h+1$　　D. $2h-1$

6. 在散列函数 $H(k)=k \bmod m$ 中,一般来讲 m 应取(　　)。

A. 奇数　　B. 偶数　　C. 素数　　D. 充分大的数

7. 静态查找与动态查找的根本区别在于(　　)。

A. 它们的逻辑结构不一样　　B. 施加在其上的操作不同

C. 所包含的数据元素的类型不一样　　D. 存储实现不一样

8. 长度为 12 的有序表采用顺序存储结构和折半查找技术,在等概率情况下查找成功时的平均查找长度是(　　),查找失败时的平均查找长度是(　　)。

A. 37/12　　B. 62/13　　C. 39/12　　D. 49/13

9. 用 n 个键值构造一棵二叉排序树,其最低高度为(　　)。

A. $n/2$　　B. n　　C. $\log_2 n$　　D. $\log_2 n+1$

10. 在二叉排序树中最小值节点的(　　)。

A. 左指针一定为空　　B. 右指针一定为空

C. 左、右指针均为空　　D. 左、右指针均不为空

11. 散列技术中的冲突指的是(　　)。

A. 两个元素具有相同的序号

B. 两个元素的关键字值不同,而其他属性相同

C. 数据元素过多

D. 不同关键字值的元素对应相同的存储地址

12. 在采用线性探测法处理冲突所构成的闭散列表上进行查找可能要探测多个位置,在查找成功的情况下所探测的这些位置的键值(　　)。

A. 一定都是同义词　　B. 一定都不是同义词

C. 不一定都是同义词　　　　　　D. 都相同

二、填空题

1. 评价查找效率的主要标准是________。

2. 查找表的逻辑结构是________。

3. 对于长度为 100 的顺序表,在等概率情况下查找成功时的平均查找长度为________,查找不成功时的平均查找长度为________。

4. 在有 150 个节点的有序表中进行二分法查找,不论成功与否,键值的比较次数最多为________。

5. 索引顺序表上的查找分两个阶段,即________、________。

6. 从 n 个节点的二叉排序树中查找一个元素,平均时间复杂度大致为________。

7. 散列表中的同义词是指________。

8. 散列表既是一种________方式,又是一种________方法。

9. 散列表中要解决的两个主要问题是________、________。

10. 散列表的冲突处理方法有________和________两种,对应的散列表分别称为开散列表和闭散列表。

三、算法设计题

1. 编写一个非递归算法,在稀疏有序索引表中二分查找出给定值 k 所对应的索引项,返回该索引项的 start 域的值。

2. 编写一个算法,在二叉排序树中查找键值为 k 的节点。

3. 设计一个简单的学生信息管理系统,每个学生的信息包括学号、姓名、性别、班级和电话等。采用二叉排序树结构实现以下功能。

(1) 创建学生的信息表。

(2) 按照学号和姓名查找学生的信息。

附录 A　数据结构试卷

数据结构试卷(一)

一、选择题(每题 2 分,共 20 分)

1. 栈和队列的共同特点是(　　)。

A. 只允许在端点处插入和删除元素

B. 都是先进后出

C. 都是先进先出

D. 没有共同点

2. 用链接方式存储的队列在进行插入运算时(　　)。

A. 仅修改头指针　　B. 头、尾指针都要修改

C. 仅修改尾指针　　D. 头、尾指针可能都要修改

3. 以下数据结构中(　　)是非线性结构。

A. 队列　　B. 栈　　C. 线性表　　D. 二叉树

4. 设有一个二维数组 A[m][n],假设 A[0][0]的存放位置在 $644_{(10)}$,A[2][2]的存放位置在 $676_{(10)}$,每个元素占一个空间,那么 $A[3][3]_{(10)}$ 存放在(　　)位置。脚注$_{(10)}$表示用十进制表示。

A. 688　　B. 678　　C. 692　　D. 696

5. 树最适合用来表示(　　)。

A. 有序数据元素　　B. 无序数据元素

C. 元素之间具有分支层次关系的数据　　D. 元素之间无联系的数据

6. 二叉树的第 k 层的节点数最多为(　　)。

A. 2^k-1　　B. $2k+1$　　C. $2k-1$　　D. 2^{k-1}

7. 若有 18 个元素的有序表存放在一维数组 A[19]中,第一个元素放在 A[1]中,现进行二分查找,则查找 A[3]的比较序列的下标依次为(　　)。

A. 1,2,3　　B. 9,5,2,3

C. 9,5,3　　D. 9,4,2,3

8. 对 n 个记录的文件进行快速排序所需要的辅助存储空间大致为(　　)。

A. $O(1)$　　B. $O(n)$　　C. $O(\log_2 n)$　　D. $O(n^2)$

9. 对线性表{7,34,55,25,64,46,20,10}进行散列存储时,若选用 $H(K)=K\%9$ 作为散列函数,则散列地址为 1 的元素有(　　)个。

A. 1　　B. 2　　C. 3　　D. 4

10. 设有 6 个节点的无向图,该图至少应有(　　)条边才能确保是一个连通图。

A. 5　　B. 6　　C. 7　　D. 8

二、填空题(每空 1 分,共 26 分)

1. 通常从 4 个方面评价算法的质量,即________、________、________和________。

2. 一个算法的时间复杂度为$(n^3+n^2\log_2 n+14n)/n^2$,其数量级表示为________。

3. 假定一棵树的广义表表示为 A(C,D(E,F,G),H(I,J)),则树中所含的节点数为________个,树的深度为________,树的度为________。

4. 若用链表存储一棵二叉树,每个节点除数据域外还有指向左孩子和右孩子的两个指针。在这种存储结构中,n 个节点的二叉树共有________个指针域,其中有个________指针域存放了地址,有________个指针是空指针。

5. 对于一个有 n 个顶点和 e 条边的有向图和无向图,在其对应的邻接表中所含的边节点分别为________个和________个。

6. AOV 网是一种________的图。

7. 在一个有 n 个顶点的无向完全图中包含有________条边,在一个有 n 个顶点的有向完全图中包含有________条边。

8. 假定一个线性表为{12,23,74,55,63,40},若按 key % 4 条件进行划分,使得同一余数的元素成为一个子表,则得到的 4 个子表分别为________、________、________和________。

9. 在向一棵 B⁻ 树插入元素的过程中若最终引起树根节点的分裂,则新树比原树的高度________。

10. 在堆排序的过程中,对任一分支节点进行筛运算的时间复杂度为________,整个堆排序过程的时间复杂度为________。

11. 在快速排序、堆排序、归并排序中________排序是稳定的。

三、计算题(每题 6 分,共 24 分)

1. 在如图 A.1 所示的数组 A 中链接存储了一个线性表,表头指针为 A[0].next,试写出该线性表。

2. 请画出图 A.2 所示的邻接矩阵和邻接表。

	0	1	2	3	4	5	6	7
data		60	50	78	90	34		40
next	3	5	7	2	0	4		1

图 A.1　数组 A

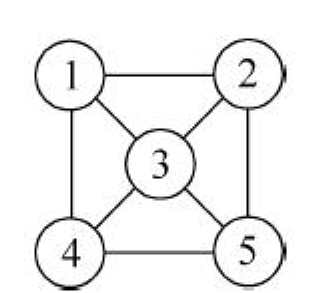


图 A.2　无向图

3. 已知一个图的顶点集 V 和边集 E 如下:

$V=\{1,2,3,4,5,6,7\}$;

$E=\{(1,2)3,(1,3)5,(1,4)8,(2,5)10,(2,3)6,(3,4)15,$

(3,5)12,(3,6)9,(4,6)4,(4,7)20,(5,6)18,(6,7)25}；

用克鲁斯卡尔算法得到最小生成树，试写出在最小生成树中依次得到的各条边。

4. 画出向小根堆中加入数据 4,2,5,8,3 时每加入一个数据后堆的变化。

四、阅读算法(每题 7 分，共 14 分)

1.

```
def mynote(L):
    # L是不带头节点的单链表的头指针
    if L is not None and L.next is not None:
        q = L
        l = L.next
        p = L
        while p.next:
            p = p.next # S1
        p.next = q
        q.next = None
    return L
```

请回答下列问题：

(1) 说明语句 S1 的功能；

(2) 说明语句组 S2 的功能；

(3) 设链表表示的线性表为(a_1，a_2，…，a_n)，写出算法执行后的返回值所表示的线性表。

2.

```
def ABC(BT):
    # BT是二叉树的节点
    if BT is not None:
        ABC(BT.lchild)
        ABC(BT.rchild)
        print(BT.data,end = ' ')
```

该算法的功能是________。

五、算法填空(共 8 分)

二叉搜索树的查找——递归算法：

```
def Find(BST,item):
    # BST是搜索二叉树的节点,item是查找的元素
    if BST is None:
        return false # 查找失败
    if item == BST.data:
        item = BST.data # 查找成功
        return ______
    elif item < BST.data:
        return Find(______,item)
```

```
        else:
            return Find(______,item)
```

六、编写算法(共 8 分)

统计出单链表 HL 中节点的值等于给定值 X 的节点数。

数据结构试卷(二)

一、选择题(每题 3 分,共 24 分)

1. 下面关于线性表的叙述错误的是(　　)。

A. 线性表采用顺序存储必须占用一片连续的存储空间
B. 线性表采用链式存储不必占用一片连续的存储空间
C. 线性表采用链式存储便于插入和删除操作的实现
D. 线性表采用顺序存储便于插入和删除操作的实现

2. 设哈夫曼树中的叶子节点总数为 m,若用二叉链表作为存储结构,则该哈夫曼树中总共有(　　)个空指针域。

A. $2m-1$　　B. $2m$　　C. $2m+1$　　D. $4m$

3. 设顺序循环队列 Q[0：M−1]的头指针和尾指针分别为 F 和 R,头指针 F 总是指向队头元素的前一位置,尾指针 R 总是指向队尾元素的当前位置,则该循环队列中的元素个数为(　　)。

A. R−F　　B. F−R
C. (R−F+M)%M　　D. (F−R+M)%M

4. 设某棵二叉树的中序遍历序列为 ABCD、前序遍历序列为 CABD,则后序遍历该二叉树得到的序列为(　　)。

A. BADC　　B. BCDA　　C. CDAB　　D. CBDA

5. 设某完全无向图中有 n 个顶点,则该完全无向图中有(　　)条边。

A. $n(n-1)/2$　　B. $n(n-1)$　　C. n^2　　D. n^2-1

6. 设某棵二叉树中有 2000 个节点,则该二叉树的最小高度为(　　)。

A. 9　　B. 10　　C. 11　　D. 12

7. 设某有向图中有 n 个顶点,则该有向图对应的邻接表中有(　　)个表头节点。

A. $n-1$　　B. n　　C. $n+1$　　D. $2n-1$

8. 设有一组初始记录关键字序列{5,2,6,3,8},以第一个记录关键字 5 为基准进行一趟快速排序的结果为(　　)。

A. 2,3,5,8,6　　B. 3,2,5,8,6
C. 3,2,5,6,8　　D. 2,3,6,5,8

二、填空题(每空 2 分,共 24 分)

1. 为了能有效地应用 Hash 查找技术,必须解决的两个问题是________。

2. 下面程序段的功能实现数据 x 进栈，要求在下画线处填上正确的语句。

```
class SqStack:
    def __init__(self):
        self.data = [None] * 100
        self.top = 0
        # ……

    # ……

    def push(self,x):
        if self.top == 100:
            raise('overflow')
        else:
            ________
            ________
```

3. 中序遍历二叉排序树所得到的序列是________序列(填有序或无序)。

4. 快速排序的最坏时间复杂度为________，平均时间复杂度为________。

5. 设某棵二叉树中度数为 0 的节点数为 N_0，度数为 1 的节点数为 N_1，则该二叉树中度数为 2 的节点数为________；若采用二叉链表作为该二叉树的存储结构，则该二叉树中共有________个空指针域。

6. 设某无向图中的顶点数和边数分别为 n 和 e，所有顶点的度数之和为 d，则 e=________。

7. 设一组初始记录关键字序列为{55,63,44,38,75,80,31,56}，则利用筛选法建立的初始堆为________。

8. 已知一个有向图的邻接表存储结构如图 A.3 所示，从顶点 1 出发，DFS 遍历的输出序列是________，BFS 遍历的输出序列是________。

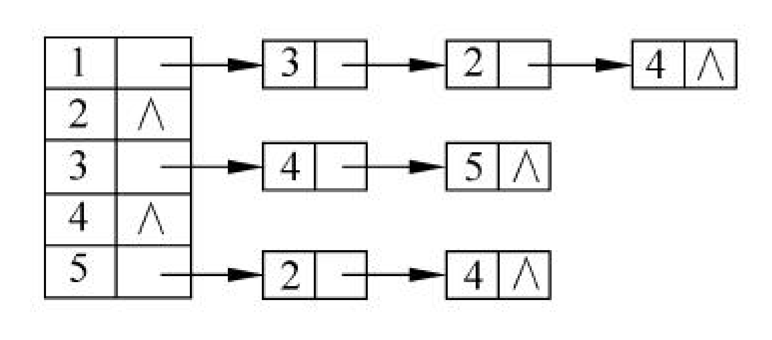


图 A.3 图的邻接表存储结构

三、应用题(每题 6 分，共 36 分)

1. 设一组初始记录关键字序列为{45,80,48,40,22,78}，则分别给出第 4 趟简单选择排序和第 4 趟直接插入排序后的结果。

2. 设指针变量 p 指向双向链表中的节点 A，指针变量 q 指向被插入节点 B，要求给出在节点 A 的后面插入节点 B 的操作序列(设双向链表中节点的两个指针域分别为 llink 和 rlink)。

3. 设一组有序的记录关键字序列为{13,18,24,35,47,50,62,83,90}，查找方法用二分查找，要求计算出查找关键字 62 时的比较次数并计算出查找成功时的平均查找长度。

4. 设一棵树 T 中边的集合为{(A,B),(A,C),(A,D),(B,E),(C,F),(C,G)}，要求用孩子兄弟表示法(二叉链表)表示出该树的存储结构并将该树转化成对应的二叉树。

5. 设有如图 A.4 所示的无向图 G，要求给出用普里姆算法构造最小生成树所走过的边的集合。

6. 设有一组初始记录关键字为{45,80,48,40,22,78}，要求构造一棵二叉排序树并给

出构造过程。

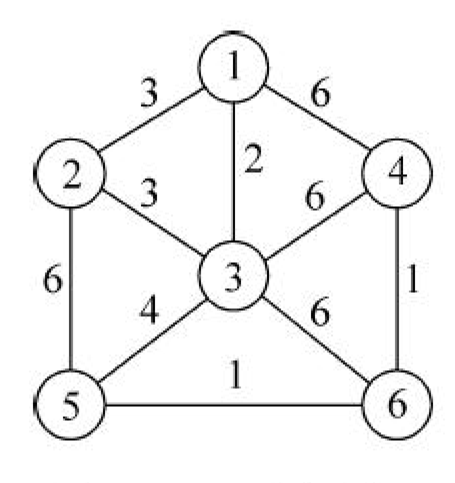


图 A.4　无向图 G

四、算法设计题(每题 8 分,共 16 分)

1. 设有一组初始记录关键字序列$\{K_1,K_2,\cdots,K_n\}$,要求设计一个算法能够在$O(n)$的时间复杂度内将线性表划分成两部分,其中左半部分的每个关键字均小于K_i,右半部分的每个关键字均大于等于K_i。

2. 设有两个集合A和集合B,要求设计生成集合$C=A\cap B$的算法,其中集合A、B和C用链式存储结构表示。

数据结构试卷(三)

一、选择题(每题 2 分,共 20 分)

1. 设某数据结构的二元组形式表示为$A=(D,R)$,$D=\{01,02,03,04,05,06,07,08,09\}$,$R=\{r\}$,$r=\{<01,02>,<01,03>,<01,04>,<02,05>,<02,06>,<03,07>,<03,08>,<03,09>\}$,则数据结构$A$是(　　)。

A. 线性结构　　B. 树结构　　C. 物理结构　　D. 图形结构

2. 下面程序的时间复杂度为(　　)。

```
i= 1
s= 0
while i<=n:
    i += 1
    t = 1
    for j in range(1,i):
        t = t * j
        s = s + t
```

A. $O(n)$　　B. $O(n^2)$　　C. $O(n^3)$　　D. $O(n^4)$

3. 设指针变量 p 指向单链表中的节点A,若删除单链表中的节点A,则需要修改指针的操作序列为(　　)。

A. q=p. next；p. data=q. data；p. next=q. next；

B. q=p. next；q. data=p. data；p. next=q. next；

C. q=p. next；p. next=q. next；

D. q=p. next；p. data=q. data；

4. 设有n个待排序的记录关键字,在堆排序中需要(　　)个辅助记录单元。

A. 1　　B. n　　C. $n\log_2 n$　　D. n^2

5. 设一组记录关键字为{20,15,14,18,21,36,40,10},则以 20 为基准记录的一趟快速排序结束后的结果为(　　)。

A. 10,15,14,18,20,36,40,21　　B. 10,15,14,18,20,40,36,21

C. 10,15,14,20,18,40,36,21　　D. 15,10,14,18,20,36,40,21

6. 设二叉排序树中有 n 个节点，则二叉排序树的平均查找长度为(　　)。

A. $O(1)$　　B. $O(\log_2 n)$　　C. $O(n)$　　D. $O(n^2)$

7. 设无向图 G 中有 n 个顶点、e 条边，则其对应的邻接表中的表头节点和表节点的个数分别为(　　)。

A. n、e　　B. e、n　　C. $2n$、e　　D. n、$2e$

8. 设某强连通图中有 n 个顶点，则该强连通图中至少有(　　)条边。

A. $n(n-1)$　　B. $n+1$　　C. n　　D. $n(n+1)$

9. 设有 5000 个待排序的记录关键字，如果需要用最快的方法选出其中最小的 10 个记录关键字，则用下列(　　)方法可以达到此目的。

A. 快速排序　　B. 堆排序　　C. 归并排序　　D. 插入排序

10. 下列 4 种排序中(　　)的空间复杂度最大。

A. 插入排序　　B. 冒泡排序　　C. 堆排序　　D. 归并排序

二、填空题(每空 1 分，共 20 分)

1. 数据的物理结构主要包括________和________两种情况。

2. 设一棵完全二叉树中有 500 个节点，则该二叉树的深度为________；若用二叉链表作为该完全二叉树的存储结构，则共有________个空指针域。

3. 设输入序列为{1,2,3}，则经过栈的作用后可以得到________种不同的输出序列。

4. 设有向图 G 用邻接矩阵 $\mathbf{A}[n][n]$ 作为存储结构，则该邻接矩阵中第 i 行上的所有元素之和等于顶点 i 的________，第 i 列上的所有元素之和等于顶点 i 的________。

5. 设哈夫曼树中共有 n 个节点，则该哈夫曼树中有________个度数为 1 的节点。

6. 设有向图 G 中有 n 个顶点、e 条有向边，所有的顶点入度数之和为 d，则 e 和 d 的关系为________。

7. ________遍历二叉排序树中的节点可以得到一个递增的关键字序列(填先序、中序或后序)。

8. 设查找表中有 100 个元素，如果用二分查找方法查找数据元素 X，则最多需要比较________次就可以断定数据元素 X 是否在查找表中。

9. 不论是顺序存储结构的栈还是链式存储结构的栈，其入栈和出栈操作的时间复杂度均为________。

10. 设有 n 个节点的完全二叉树，如果按照从自上到下、从左到右从 1 开始顺序编号，则第 i 个节点的双亲节点的编号为________，右孩子节点的编号为________。

11. 设一组初始记录关键字为{72,73,71,23,94,16,5}，则以记录关键字 72 为基准的一趟快速排序的结果为________。

12. 设有向图 G 中的有向边的集合 $E=\{<1,2>,<2,3>,<1,4>,<4,2>,<4,3>\}$，则该图的一种拓扑序列为________。

13. 下列算法实现在顺序散列表中查找值为 x 的关键字的功能，请在下画线处填上正确的语句。

```
class record(object):
```

```
    def __init__(self,key,others):
        self.key = key
        self.others = others

def hashSqSearch(hashTable,k):
    i = j = k % P
    while hashTable[j].key!= k and hashTable[j].flag!= 0:
        j = ______ % m
        if i == j:
            return -1
    if ______:
        return j
    else return -1
```

14. 下列算法实现在二叉排序树上查找关键值 k 的功能,请在下画线处填上正确的语句。

```
def Find(BST,k):
    # BST 是搜索二叉树的节点,k 是查找的元素
    if BST is None:
        return false # 查找失败
    if k == BST.data:
        k = BST.data # 查找成功
        return ______
    elif item < BST.data:
        return Find(________,k)
    else:
        return Find(________,k)
```

三、计算题(每题 10 分,共 30 分)

1. 已知二叉树的前序遍历序列是 AEFBGCDHIKJ、中序遍历序列是 EFAGBCHKIJD,画出此二叉树,并画出它的后序线索二叉树。

2. 已知待散列的线性表为{36,15,40,63,22},散列用的一维地址空间为[0..6],假定选用的散列函数是 $H(K) = K \bmod 7$,若发生冲突采用线性探查法处理,试计算以下问题:

(1) 计算出每一个元素的散列地址并在图 A.5 中填写出散列表。

(2) 求出在查找每一个元素概率相等情况下的平均查找长度。

0	1	2	3	4	5	6

图 A.5 填写散列表

3. 已知序列{10,18,4,3,6,12,1,9,18,8},请用快速排序写出每一趟排序的结果。

四、算法设计题(每题 15 分,共 30 分)

1. 设计在单链表中删除值相同的多余节点的算法。

2. 设计一个求节点 x 在二叉树中的双亲节点的算法。

数据结构试卷(四)

一、选择题(每题 2 分,共 20 分)

1. 设一维数组中有 n 个数组元素,则读取第 i 个数组元素的平均时间复杂度为(　　)。

A. $O(n)$　　B. $O(n\log_2 n)$　　C. $O(1)$　　D. $O(n^2)$

2. 设一棵二叉树的深度为 k,则该二叉树中最多有(　　)个节点。

A. $2k-1$　　B. 2^k　　C. 2^{k-1}　　D. 2^k-1

3. 设某无向图中有 n 个顶点、e 条边,则该无向图中所有顶点的入度之和为(　　)。

A. n　　B. e　　C. $2n$　　D. $2e$

4. 在二叉排序树中插入一个节点的时间复杂度为(　　)。

A. $O(1)$　　B. $O(n)$　　C. $O(\log_2 n)$　　D. $O(n^2)$

5. 设某有向图的邻接表中有 n 个表头节点和 m 个表节点,则该图中有(　　)条有向边。

A. n　　B. $n-1$　　C. m　　D. $m-1$

6. 设一组初始记录关键字序列为{345,253,674,924,627},则用基数排序需要进行(　　)趟的分配和回收才能使初始关键字序列变成有序序列。

A. 3　　B. 4　　C. 5　　D. 8

7. 设用链表作为栈的存储结构,则退栈操作(　　)。

A. 必须判别栈是否为满　　B. 必须判别栈是否为空

C. 判别栈元素的类型　　D. 对栈不做任何判别

8. 下列 4 种排序中(　　)的空间复杂度最大。

A. 快速排序　　B. 冒泡排序　　C. 希尔排序　　D. 堆

9. 设某二叉树中度数为 0 的节点数为 N_0,度数为 1 的节点数为 N_1,度数为 2 的节点数为 N_2,则下列等式成立的是(　　)。

A. $N_0=N_1+1$　　B. $N_0=N_1+N_2$

C. $N_0=N_2+1$　　D. $N_0=2N_1+1$

10. 设有序顺序表中有 n 个数据元素,则利用二分查找法查找数据元素 X 的最多比较次数不超过(　　)。

A. $\log_2 n+1$　　B. $\log_2 n-1$　　C. $\log_2 n$　　D. $\log_2(n+1)$

二、填空题(除第 2 题 2 分外每空 1 分,共 20 分)

1. 设有 n 个无序的记录关键字,则直接插入排序的时间复杂度为________,快速排序的平均时间复杂度为________。

2. 设指针变量 p 指向双向循环链表中的节点 X,则删除节点 X 需要执行的语句序列为________(设节点中的两个指针域分别为 llink 和 rlink)。

3. 根据初始关键字序列{19,22,01,38,10}建立的二叉排序树的高度为________。

4. 深度为 k 的完全二叉树中最少有________个节点。

5. 设初始记录关键字序列为$\{K_1, K_2, \cdots, K_n\}$,则用筛选法思想建堆必须从第________个元素开始进行筛选。

6. 设哈夫曼树中共有 99 个节点,则该树中有个叶子节点;若采用二叉链表作为存储结构,则该树中有________个空指针域。

7. 设一个顺序循环队列中有 M 个存储单元,则该循环队列中最多能够存储________个队列元素;当前实际存储________个队列元素(设头指针 F 指向当前队头元素的前一个位置,尾指针指向当前队尾元素的位置)。

8. 设顺序线性表中有 n 个数据元素,则在第 i 个位置上插入一个数据元素需要移动表中的________个数据元素;删除第 i 个位置上的数据元素需要移动表中的________个元素。

9. 设一组初始记录关键字序列为{20,18,22,16,30,19},则以 20 为中轴的一趟快速排序的结果为________。

10. 设一组初始记录关键字序列为{20,18,22,16,30,19},则根据这些初始关键字序列建成的初始堆为________。

11. 设某无向图 G 中有 n 个顶点,用邻接矩阵 $\boldsymbol{A}$ 作为该图的存储结构,则顶点 i 和顶点 j 互为邻接点的条件是________。

12. 设无向图对应的邻接矩阵为 $\boldsymbol{A}$,则 $\boldsymbol{A}$ 中第 i 行上非 0 元素的个数________第 i 列上非 0 元素的个数(填等于、大于或小于)。

13. 设前序遍历某二叉树的序列为 ABCD,中序遍历该二叉树的序列为 BADC,则后序遍历该二叉树的序列为________。

14. 设散列函数 $H(k)=k \bmod p$,解决冲突的方法为链地址法。要求在下列算法画线处填上正确的语句完成在散列表 hashtalbe 中查找关键字值等于 k 的节点,成功时返回指向关键字的指针,不成功时返回标志 0。

```
class Node(object):
    def __init__(self,key = None,next = None):
        self.key = key
        self.next = next

def createlkHash(hashTable):
    for i in range(m):
        ________
    for i in range(n):
        s = Node()
        s.key = a[i]
        k = a[i] % P
        s.next = hashTable[k]
```

三、计算题(每题 10 分,共 30 分)

1. 画出广义表 LS=(( ),(e),(a,(b,c,d)))的头尾链表存储结构。

2. 设有如图 A.6 所示的森林:

(1) 求树(a)的先根序列和后根序列;

(2) 求森林的先序序列和中序序列;

(3) 将此森林转换为相应的二叉树。

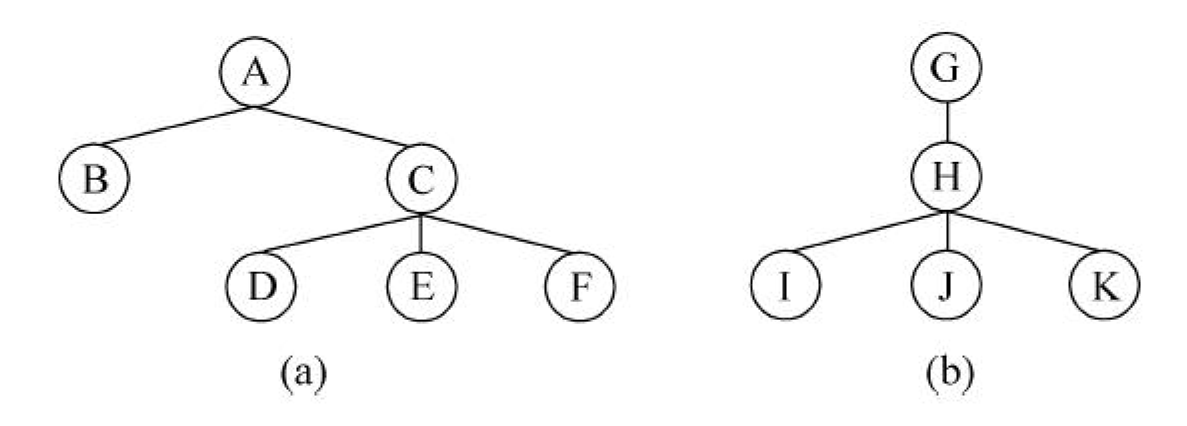


图 A.6　森林图

3. 设散列表的地址范围是[0..9],散列函数为 $H(\text{key}) = (\text{key}^2 + 2) \bmod 9$,并采用链表处理冲突,请画出元素 7,4,5,3,6,2,8,9 依次插入散列表的存储结构。

四、算法设计题(每题 10 分,共 30 分)

1. 设单链表中仅有 3 类字符的数据元素(大写字母、数字和其他字符),要求利用原单链表中的节点空间设计出 3 个单链表的算法,使每个单链表只包含同类字符。

2. 设计在链式存储结构上交换二叉树中所有节点左、右子树的算法。

3. 在链式存储结构上建立一棵二叉排序树。

数据结构试卷(五)

一、选择题(每题 2 分,共 20 分)

1. 数据的最小单位是(　　)。

A. 数据项　　B. 数据类型　　C. 数据元素　　D. 数据变量

2. 设一组初始记录关键字序列为{50,40,95,20,15,70,60,45},则以增量 $d=4$ 的一趟希尔排序结束后前 4 条记录关键字为(　　)。

A. 40,50,20,95　　B. 15,40,60,20

C. 15,20,40,45　　D. 45,40,15,20

3. 设一组初始记录关键字序列为{25,50,15,35,80,85,20,40,36,70},其中含有 5 个长度为 2 的有序子表,则用归并排序的方法对该记录关键字序列进行一趟归并后的结果为(　　)。

A. 15,25,35,50,20,40,80,85,36,70　　B. 15,25,35,50,80,20,85,40,70,36

C. 15,25,35,50,80,85,20,36,40,70　　D. 15,25,35,50,80,20,36,40,70,85

4. 函数 substr("DATASTRUCTURE",5,9)的返回值为(　　)。

A. "STRUCTURE"　　B. "DATA"

C. "ASTRUCTUR"　　D. "DATASTRUCTURE"

5. 设一个有序的单链表中有 n 个节点,现要求插入一个新节点后使得单链表仍然保持有序,则该操作的时间复杂度为(　　)。

A. $O(\log_2 n)$　　B. $O(1)$　　C. $O(n^2)$　　D. $O(n)$

6. 设一棵 m 叉树中度数为 0 的节点数为 N_0,度数为 1 的节点数为 N_1,…,度数为 m 的

节点数为 N_m,则 N_0=(   )。

A. $N_1+N_2+\cdots+N_m$ B. $1+N_2+2N_3+3N_4+\cdots+(m-1)N_m$

C. $N_2+2N_3+3N_4+\cdots+(m-1)N_m$ D. $2N_1+3N_2+\cdots+(m+1)N_m$

7. 设有序表中有 1000 个元素,则用二分查找法查找元素 X 最多需要比较(   )次。

A. 25 B. 10 C. 7 D. 1

8. 设连通图 G 中的边集 $E=\{(a,b),(a,e),(a,c),(b,e),(e,d),(d,f),(f,c)\}$,则从顶点 a 出发可以得到一种深度优先遍历的顶点序列为(   )。

A. abedfc B. acfebd C. aebdfc D. aedfcb

9. 设输入序列是$\{1,2,3,\cdots,n\}$,经过栈的作用后输出序列的第一个元素是 n,则输出序列中的第 i 个输出元素是(   )。

A. $n-i$ B. $n-1-i$ C. $n+1-i$ D. 不能确定

10. 设一组初始记录关键字序列为{45,80,55,40,42,85},则以第一个记录关键字 45 为基准得到的一趟快速排序的结果是(   )。

A. 40,42,45,55,80,83 B. 42,40,45,80,85,88

C. 42,40,45,55,80,85 D. 42,40,45,85,55,80

二、填空题(除第 1、2、8 题 2 分外每空 1 分,共 20 分)

1. 设有一个顺序共享栈 S[0: $n-1$],其中第一个栈顶指针 top1 的初值为 -1,第二个栈顶指针 top2 的初值为 n,则判断共享栈满的条件是________。

2. 在图的邻接表中用顺序存储结构存储表头节点的优点是________。

3. 设有一个 n 阶的下三角矩阵 $\mathbf{A}$,如果按照行的顺序将下三角矩阵中的元素(包括对角线上的元素)存放在 $n(n+1)$个连续的存储单元中,则 A[i][j]与 A[0][0]之间有________个数据元素。

4. 栈的插入和删除只能在栈的栈顶进行,后进栈的元素必定先出栈,所以又把栈称为________表;队列的插入和删除运算分别在队列的两端进行,先进队列的元素必定先出队列,所以又把队列称为________表。

5. 设一棵完全二叉树的顺序存储结构中的存储数据元素为 ABCDEF,则该二叉树的前序遍历序列为________、中序遍历序列为________、后序遍历序列为________。

6. 设一棵完全二叉树有 128 个节点,则该完全二叉树的深度为________,有________个叶子节点。

7. 设有向图 G 的存储结构用邻接矩阵 $\mathbf{A}$ 来表示,则 $\mathbf{A}$ 中第 i 行中的所有非零元素个数之和等于顶点 i 的________,第 i 列中的所有非零元素个数之和等于顶点 i 的________。

8. 设一组初始记录关键字序列($k_1,k_2,\cdots,k_n$)是堆,则对 $i=1,2,\cdots,n/2$ 而言满足的条件为________。

9. 下面程序段的功能是实现冒泡排序算法,请在下画线处填上正确的语句。

```
def bubbleSort(sqlist):
    flag = True
    i = 1
    while i < sqlist.len and flag:
```

```
        flag = False
        for j in range(______:
            if sqlist.list[j+1].key < sqlist.list[j].key:
                p = sqlist.list[j]
                ______
                sqlist.list[j+1] = p
                flag = True
        i += 1
```

10. 下面程序段的功能是实现二分查找算法，请在下画线处填上正确的语句。

```
class record(object):
    def __init__(self,key=None,other=None):
        self.key = key
        self.other = other

def bisearch(r,k):
    low = 0
    high = len(r)-1
    while low<=high:
        ______
        if r[mid].key == k:
            return mid+1
        elif ______:
            high = mid-1
        else:
            low = mid+1
    return -1
```

三、应用题(每题 8 分，共 32 分)

1. 设某棵二叉树的中序遍历序列为 DBEAC、前序遍历序列为 ABDEC，要求给出该二叉树的后序遍历序列。

2. 设有无向图 G(如图 A.7 所示)，给出该图的最小生成树上边的集合，并计算最小生成树各边上的权值之和。

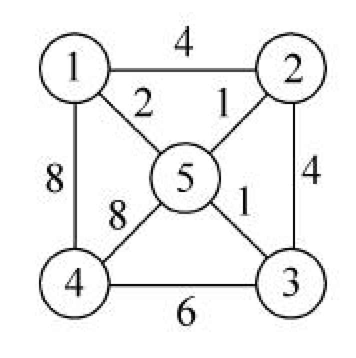


图 A.7　无向图 G

3. 设一组初始记录关键字序列为{15,17,18,22,35,51,60}，要求计算出成功查找时的平均查找长度。

4. 设散列表的长度为 8，散列函数 $H(k)=k \bmod 7$，初始记录关键字序列为{25,31,8,27,13,68}，要求分别计算出用线性探测法和链地址法作为解决冲突方法的平均查找长度。

四、算法设计题(每题 14 分，共 28 分)

1. 设计判断两个二叉树是否相同的算法。
2. 设计两个有序单链表的合并排序算法。

附录B 实 践 题

第2章 线 性 表

1. 编写一组程序，基于单链表，用头插法建表，实现某班学生姓名数据的建表、展示、查找、定位、插入、删除、判定表空、求表长等操作。

依次输入学生姓名：

赵壹、钱贰、孙叁、李肆、周伍、吴陆、郑柒、王捌

实验测试要求如下：

(1) 展示该班所有学生的姓名及班级人数。

(2) 查找学生“李肆”在表中的位置。

(3) 在表中的学生“王捌”后加入新生“冯玖”，删除班里的转走生“赵壹”，展示该班的现有学生。

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IList(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将线性表置成空表'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断线性表是否为空表'''
        pass
    @abstractmethod
    def length(self):
        '''返回线性表的长度'''
        pass
    @abstractmethod
    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        pass
    @abstractmethod
    def insert(self,i,x):
        '''插入 x 作为第 i 个元素'''
        pass
    @abstractmethod
```

```
    def remove(self,i):
        '''删除第 i 个元素'''
        pass
    @abstractmethod
    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        pass
    @abstractmethod
    def display(self):
        '''输出线性表中各个数据元素的值'''
        pass

class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next

class LinkList(IList):
    def __init__(self):
        self.head = Node() # 构造函数初始化头节点

    def create(self,l,order):
        if order:
            self.create_tail(l)
        else:
            self.create_head(l)

    def create_tail(self,l):
        for item in l:
            self.insert(self.length(),item)

    def create_head(self,l):
        for item in l:
            self.insert(0,item)

    def clear(self):
        '''将线性表置成空表'''
        self.head.data = None
        self.head.next = None

    def isEmpty(self):
        '''判断线性表是否为空表'''
        return self.head.next == None

    def length(self):
        '''返回线性表的长度'''
        p = self.head.next
        length = 0
        while p is not None:
            p = p.next
```

```
            length += 1
        return length

    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        p = self.head.next # p 指向单链表的首节点
        j = 0
        # 从首节点开始向后查找,直到 p 指向第 i 个节点或者 p 为 null
        while j < i and p is not None:
            p = p.next
            j += 1
        if j > i or p is None: # i 不合法时抛出异常
            raise Exception("第" + i + "个数据元素不存在")
        return p.data

    def insert(self,i,x):
        '''(带头节点)插入 x 作为第 i 个元素'''
        p = self.head
        j = -1
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p is None:
            raise Exception("插入位置不合法")
        s = Node(x,p.next)
        p.next = s

    '''
    def insert(self,i,x):
        # (不带头节点)插入 x 作为第 i 个元素
        p = self.head
        j = -1
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p is None:
            raise Exception("插入位置不合法")
        s = Node(data = x)
        if i == 0:
            s.next = self.head
        else:
            s.next = p.next
            p.next = s
    '''

    def remove(self,i):
        '''删除第 i 个元素'''
        p = self.head
        j = -1
        # 寻找第 i 个节点的前驱节点
```

```
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p.next is None:
            raise Exception("删除位置不合法")
        p.next = p.next.next

    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        p = self.head.next
        j = 0
        while p is not None and not (p.data == x):
            p = p.next
            j += 1
        if p is not None:
            return j
        else:
            return -1

    def display(self):
        '''输出线性表中各个数据元素的值'''
        p = self.head.next
        while p is not None:
            print(p.data,end = ' ')
            p = p.next

L = LinkList()
for i in range(8):
    s = input()
    L.insert(i,s)
# 依次输入：赵壹、钱贰、孙叁、李肆、周伍、吴陆、郑柒、王捌
# 注意每输入一个人名后键入一次回车
# (1)
print("(1)班级学生：",end = '')
L.display()
print("班级人数：",end = '')
print(L.length())
# (2)
print("(2)'李肆'在表中的位置:",L.indexOf("李肆"))
# (3)
L.insert(L.indexOf("王捌") + 1,"冯玖")
L.remove(L.indexOf("赵壹"))
print("(3)现在该班级学生：",end = '')
L.display()
print()
```

代码输出：

```
(1)班级学生：赵壹 钱贰 孙叁 李肆 周伍 吴陆 郑柒 王捌 班级人数：8
(2)'李肆'在表中的位置：3
(3)现在该班级学生：钱贰 孙叁 李肆 周伍 吴陆 郑柒 王捌 冯玖
```

2. 编写一组程序,基于单链表,实现一元多项式的加法运算。

多项式加法举例:

p1=3x^3+5x^2+4x

p2=x^5+3x^2

p1+p2=x^5+3x^3+8x^2+4x

输入:从大到小依次输入所要输入的两个一元多项式的系数和指数

输出:输出一元多项式 p1、p2 以及两式相加的结果

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IList(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将线性表置成空表'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断线性表是否为空表'''
        pass
    @abstractmethod
    def length(self):
        '''返回线性表的长度'''
        pass
    @abstractmethod
    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        pass
    @abstractmethod
    def insert(self,i,x):
        '''插入 x 作为第 i 个元素'''
        pass
    @abstractmethod
    def remove(self,i):
        '''删除第 i 个元素'''
        pass
    @abstractmethod
    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        pass
    @abstractmethod
    def display(self):
        '''输出线性表中各个数据元素的值'''
        pass

class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next
```

```python

class LinkList(IList):
    def __init__(self):
        self.head = Node() # 构造函数初始化头节点

    def create(self,l,order):
        if order:
            self.create_tail(l)
        else:
            self.create_head(l)

    def create_tail(self,l):
        for item in l:
            self.insert(self.length(),item)

    def create_head(self,l):
        for item in l:
            self.insert(0,item)

    def clear(self):
        '''将线性表置成空表'''
        self.head.data = None
        self.head.next = None

    def isEmpty(self):
        '''判断线性表是否为空表'''
        return self.head.next == None

    def length(self):
        '''返回线性表的长度'''
        p = self.head.next
        length = 0
        while p is not None:
            p = p.next
            length += 1
        return length

    def get(self,i):
        '''读取并返回线性表中的第 i 个数据元素'''
        p = self.head.next # p 指向单链表的首节点
        j = 0
        # 从首节点开始向后查找,直到 p 指向第 i 个节点或者 p 为 null
        while j < i and p is not None:
            p = p.next
            j += 1
        if j > i or p is None: # i 不合法时抛出异常
            raise Exception("第" + i + "个数据元素不存在")
        return p.data

    def insert(self,i,x):
```

```
        '''(带头节点)插入 x 作为第 i 个元素'''
        p = self.head
        j = -1
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p is None:
            raise Exception("插入位置不合法")
        s = Node(x,p.next)
        p.next = s

    '''
    def insert(self,i,x):
        # (不带头节点)插入 x 作为第 i 个元素
        p = self.head
        j = -1
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p is None:
            raise Exception("插入位置不合法")
        s = Node(data = x)
        if i == 0:
            s.next = self.head
        else:
            s.next = p.next
            p.next = s
    '''

    def remove(self,i):
        '''删除第 i 个元素'''
        p = self.head
        j = -1
        # 寻找第 i 个节点的前驱节点
        while p is not None and j < i - 1:
            p = p.next
            j += 1
        if j > i - 1 or p.next is None:
            raise Exception("删除位置不合法")
        p.next = p.next.next

    def indexOf(self,x):
        '''返回元素 x 首次出现的位序号'''
        p = self.head.next
        j = 0
        while p is not None and not (p.data == x):
            p = p.next
            j += 1
        if p is not None:
            return j
```

```
        else:
            return -1

    def display(self):
        '''输出线性表中各个数据元素的值'''
        p = self.head.next
        while p is not None:
            print(p.data,end = ' ')
            p = p.next

class PloyNode(object):
    def __init__(self,a,i):
        self.a = a # 系数
        self.i = i # 指数

def add(p1,p2):
    L = LinkList()
    i = j = 0
    while i < p1.length() and j < p2.length():
        x,y = p1.get(i),p2.get(j)
        if x.i == y.i:
            L.insert(L.length(),PloyNode(x.a + y.a,x.i))
            i += 1
            j += 1
        elif x.i > y.i:
            L.insert(L.length(),PloyNode(x.a,x.i))
            i += 1
        else:
            L.insert(L.length(),PloyNode(y.a,y.i))
            j += 1
    while i < p1.length():
        x = p1.get(i)
        L.insert(L.length(),PloyNode(x.a,x.i))
        i += 1
    while j < p2.length():
        y = p2.get(j)
        L.insert(L.length(),PloyNode(y.a,y.i))
        j += 1
    return L


p1 = LinkList()
p2 = LinkList()
# 多项式 p1 : 3x^3 + 5x^2 + 4x
p1.insert(0,PloyNode(3,3))
p1.insert(1,PloyNode(5,2))
p1.insert(2,PloyNode(4,1))
# 多项式 p2 : x^5 + 3x^2
p2.insert(0,PloyNode(1,5))
p2.insert(1,PloyNode(3,2))
```

```
# 相加
L= add(p1,p2)
for i in range(L.length() - 1):
    x = L.get(i)
    print("%sx^%s+" % (x.a,x.i),end='')
x= L.get(L.length() - 1)
print("%sx^%s" % (x.a,x.i))
```

代码输出:

```
1x^5+3x^3+8x^2+4x^1
```

3. 基于双向链表的约瑟夫问题。这是一个有名的问题,N 个人围成一圈,从第一个开始报数,第 M 个玩家将出局,继续从下一个玩家开始从头报数循环,最后剩下一个。例如 $N=6$、$M=5$,依次出局的人的序号为 5、4、6、2、3。最后优胜者为剩下的 1 号。

输入:玩家数、游戏开始数字、游戏要玩的数字 M。

输出:圆桌上的所有玩家、圆桌玩家的出局顺序、优胜者的号码。

```
class JosephusNode(object):
    def __init__(self,data,next=None,prior=None):
        self.data = data       # 序号
        self.next = None       # 下一个节点
        self.prior = None      # 上一个节点

class JosephusList(object):
    def __init__(self):
        self.curLen = 0
        self.head = None

    def insert(self,x):
        if self.head is None:
            self.head = JosephusNode(x)
            self.head.next = self.head
            self.head.prior = self.head
        else:
            s = JosephusNode(x)
            s.next = self.head
            s.prior = self.head.prior
            self.head.prior.next = s
            self.head.prior = s
        self.curLen += 1

    def remove(self,x):
        p = self.head
        while True:
            if p.data == x:
                p.prior.next = p.next
                p.next.prior = p.prior
                if p is self.head:
```

```python
                    self.head = p.next
                self.curLen -= 1
                break
            p = p.next
            if p is self.head:
                break

    def display(self):
        p = self.head
        while True:
            print(p.data,end = ' ')
            p = p.next
            if p is self.head:
                break

N = 6
M = 5
J = JosephusList()
for i in range(1,N+1):
    J.insert(i)

p = J.head
while J.curLen>1:
    for i in range(M-1):
        p = p.next
    print(p.data,"出局")
    J.remove(p.data)
    p = p.next
print(p.data,"胜出")
```

代码输出：

```
5 出局
4 出局
6 出局
2 出局
3 出局
1 胜出
```

第3章　栈和队列

1. 在一个用字符串描述的表达式“{[(a+b)]/f+(c+d)}”中存在大、中、小括号。请编写一组程序，基于栈，实现对输入的一串字符串的依次扫描，并检查括号匹配是否成功。

输入样例：{{}}()(hello){({world}{})}

输出：括号匹配成功

输入样例：{{}}()(hello){({world}()}

输出：括号匹配不成功

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IStack(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将栈置空'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断栈是否为空'''
        pass
    @abstractmethod
    def length(self):
        '''返回栈的数据元素个数'''
        pass
    @abstractmethod
    def peek(self):
        '''返回栈顶元素'''
        pass
    @abstractmethod
    def push(self,x):
        '''数据元素 x 入栈'''
        pass
    @abstractmethod
    def pop(self):
        '''将栈顶元素出栈并返回'''
        pass
    @abstractmethod
    def display(self):
        '''输出栈中的所有元素'''
        pass

class SqStack(IStack):
    def __init__(self,maxSize):
        self.maxSize = maxSize # 栈的最大存储单元个数
        self.stackElem = [None] * self.maxSize # 顺序栈存储空间
        self.top = 0 # 指向栈顶元素的下一个存储单元位置

    def clear(self):
        '''将栈置空'''
        self.top = 0

    def isEmpty(self):
        '''判断栈是否为空'''
        return self.top == 0

    def length(self):
        '''返回栈的数据元素个数'''
        return self.top

```

```
    def peek(self):
        '''返回栈顶元素'''
        if not self.isEmpty():
            return self.stackElem[self.top - 1]
        else:
            return None
    def push(self,x):
        '''数据元素 x 入栈'''
        if self.top == self.maxSize:
            raise Exception("栈已满")
        self.stackElem[self.top] = x
        self.top += 1

    def pop(self):
        '''将栈顶元素出栈并返回'''
        if self.isEmpty():
            return None
        self.top -= 1
        return self.stackElem[self.top]

    def display(self):
        '''输出栈中的所有元素'''
        for i in range(self.top - 1, - 1, - 1):
            print(self.stackElem[i],end = ' ')

def isMatched(str):
    backref = {
        ')':'(',
        ']':'[',
        '}':'{',
    }
    s = SqStack(100)
    for c in str:
        if c == '(' or c == '[' or c == '{':
            s.push(c)
        if c == ')' or c == ']' or c == '}':
            if s.isEmpty():
                return False
            if s.peek() == backref[c]:
                s.pop()
            else:
                return False
    if s.isEmpty():
        return True
    else:
        return False

s1 = "{{}}()(hello){({world}{})}"
s2 = "{{}}()(hello){({world}()}"
print("s1",end = '')
```

```
if isMatched(s1):
    print("括号匹配成功")
else:
    print("括号匹配不成功")
print("s2",end='')
if isMatched(s2):
    print("括号匹配成功")
else:
    print("括号匹配不成功")
```

代码输出：

```
s1 括号匹配成功
s2 括号匹配不成功
```

2. 杨辉三角形如图 B.1 所示，它的每行每列之间存在一定的规律。请编写一组程序，基于队列，实现杨辉三角形的打印。

```
         1   1
       1   2   1
     1   3   3   1
   1   4   6   4   1
 1   5  10  10   5   1
1  6  15  20  15   6   1
```

图 B.1 杨辉三角形

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IQueue(metaclass=ABCMeta):
    @abstractmethod
    def clear(self):
        '''将队列置空'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断队列是否为空'''
        pass
    @abstractmethod
    def length(self):
        '''返回队列的数据元素个数'''
        pass
    @abstractmethod
    def peek(self):
        '''返回队首元素'''
        pass
    @abstractmethod
    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        pass
    @abstractmethod
    def poll(self):
        '''将队首元素删除并返回其值'''
        pass
    @abstractmethod
    def display(self):
        '''输出队列中的所有元素'''
```

```
        pass

class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next

class LinkQueue(IQueue):
    def __init__(self):
        self.front = None # 队首指针
        self.rear = None # 队尾指针

    def clear(self):
        '''将队列置空'''
        self.front = None
        self.rear = None

    def isEmpty(self):
        '''判断队列是否为空'''
        return self.front is None

    def length(self):
        '''返回队列的数据元素个数'''
        p = self.front
        i = 0
        while p is not None:
            p = p.next
            i += 1
        return i

    def peek(self):
        '''返回队首元素'''
        if self.isEmpty():
            return None
        else:
            return self.front.data

    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        s = Node(x,None)
        if not self.isEmpty():
            self.rear.next = s
        else:
            self.front = s
        self.rear = s

    def poll(self):
        '''将队首元素删除并返回其值'''
        if self.isEmpty():
            return None
```

```
            p = self.front
            self.front = self.front.next
            if p == self.rear: # 删除节点为队尾节点时需要修改 rear
                self.rear = None
            return p.data

    def display(self):
        '''输出队列中的所有元素'''
        p = self.front
        while p is not None:
            print(p.data,end=' ')
            p = p.next

N= 5
q= LinkQueue()
s= 0
# 先添加两个 1 进入队列
q.offer(1)
q.offer(1)
for i in range(1,N+1):
    # 各行间插入一个 0
    q.offer(0)
    for j in range(1,i+3):
        # 退出一个系数 temp
        temp = q.poll()
        # 计算下一行的系数并加入队列
        q.offer(temp+s);
        s = temp
        if(j!= i+2):
            print(s,end=' ')
    # 换行
    print()
```

代码输出：

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

第 4 章　串和数组

1. 请编写一组程序，有两个函数，计算 next 失配函数和使用 KMP 算法实施串的模式匹配。输入两个字符串 a、b，调用这两个函数进行模式匹配，如果 a 中存在字符串 b，则输出“匹配”，否则输出“不匹配”。

输入样例：

abcbcaabc

bcaa

输出：

匹配

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IString(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将字符串置成空串'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断是否为空串'''
        pass
    @abstractmethod
    def length(self):
        '''返回串的长度'''
        pass
    @abstractmethod
    def charAt(self,i):
        '''读取并返回串中的第 i 个数据元素'''
        pass
    @abstractmethod
    def subString(self,begin,end):
        '''返回位序号从 begin 到 end - 1 的子串'''
        pass
    @abstractmethod
    def insert(self,i,str):
        '''在第 i 个字符之前插入子串 str'''
        pass
    @abstractmethod
    def delete(self,begin,end):
        '''删除位序号从 begin 到 end - 1 的子串'''
        pass
    @abstractmethod
    def concat(self,str):
        '''将 str 连接到字符串的后面'''
        pass
    @abstractmethod
    def compareTo(self,str):
        '''比较 str 和当前字符串的大小'''
        pass
    @abstractmethod
    def indexOf(self,str,begin):
        '''从位序号为 begin 的字符开始搜索与 str 相等的子串'''
        pass

```

```
class SqString(IString):
    def __init__(self,obj = None):
        if obj is None: # 构造空串
            self.strValue = [] # 字符数组存放串值
            self.curLen = 0 # 当前串的长度
        elif isinstance(obj,str): # 以字符串构造串
            self.curLen = len(obj)
            self.strValue = [None] * self.curLen
            for i in range(self.curLen):
                self.strValue[i] = obj[i]
        elif isinstance(obj,list): # 以字符列表构造串
            self.curLen = len(obj)
            self.strValue = [None] * self.curLen
            for i in range(self.curLen):
                self.strValue[i] = obj[i]

    def clear(self):
        '''将字符串置成空串'''
        self.curLen = 0

    def isEmpty(self):
        '''判断是否为空串'''
        return self.curLen == 0

    def length(self):
        '''返回串的长度'''
        return self.curLen

    def charAt(self,i):
        '''读取并返回串中的第 i 个数据元素'''
        if i<0 or i>= self.curLen:
            raise IndexError("String index out of range")
        return self.strValue[i]

    def allocate(self,newCapacity):
        '''将串的长度扩充为 newCapacity'''
        tmp = self.strValue
        self.strValue = [None] * newCapacity
        for i in range(self.curLen):
            self.strValue[i] = tmp[i]

    def subString(self,begin,end):
        '''返回位序号从 begin 到 end-1 的子串'''
        if begin<0 or begin>= end or end>self.curLen:
            raise IndexError("参数不合法")
        tmp = [None] * (end-begin)
        for i in range(begin,end):
            tmp[i-begin] = self.strValue[i] # 复制子串
        return SqString(tmp)

```

```
    def insert(self,i,str):
        '''在第 i 个字符之前插入子串 str'''
        if i<0 or i>self.curLen:
            raise IndexError("插入位置不合法")
        length = str.length()
        newCapacity = self.curLen + length
        self.allocate(newCapacity)
        for j in range(self.curLen-1,i-1,-1):
            self.strValue[j+length] = self.strValue[j]
        for j in range(i,i+length):
            # print(j-i,str.charAt(j-i))
            self.strValue[j] = str.charAt(j-i)
        self.curLen = newCapacity

    def delete(self,begin,end):
        '''删除位序号从 begin 到 end-1 的子串'''
        if begin<0 or begin>=end or end>self.curLen:
            raise IndexError("参数不合法")
        for i in range(begin,end):
            self.strValue[i] = self.strValue[i+end-begin]
        self.curLen = self.curLen - end + begin

    def concat(self,str):
        '''将 str 连接到字符串的后面'''
        self.insert(self.curLen,str)

    def compareTo(self,str):
        '''比较 str 和当前字符串的大小'''
        n = self.curLen if self.curLen<str.length() else str.length()
        for i in range(n):
            if self.strValue[i]>str.charAt(i):
                return 1
            if self.strValue[i]<str.charAt(i):
                return -1
        if self.curLen>str.length():
            return 1
        elif self.curLen<str.length():
            return -1
        return 0

    def indexOf(self,str,begin):
        '''从位序号为 begin 的字符开始搜索与 str 相等的子串'''
        pass

    def BF(self,str,begin):
        count = 0
        if str.length()<=self.curLen and str is not None and self.curLen>0:
            i = begin
            length = str.length()
            while(i<=self.curLen-length):
```

```
                for j in range(length):
                    count += 1
                    if str.charAt(j)!= self.strValue[j + i]:
                        i += 1
                        break
                    elif j == length - 1:
                        return i,count
        return -1,count

    def next(p):
        next = [0] * p.length() # next 数组
        k = 0 # 模式串指针
        j = 1 # 主串指针
        next[0] = -1
        next[1] = 0
        while j < p.length() - 1:
            if p.charAt(j) == p.charAt(k):
                next[j + 1] = k + 1
                k += 1
                j += 1
            elif k == 0:
                next[j + 1] = 0
                j += 1
            else:
                k = next[k]
        return next

    def KMP(self,p,begin):
        count = 0
        next = SqString.next(p) # 计算 next 值
        i = begin # # i 为主串的字符指针
        j = 0
        while i < self.curLen and j < p.length():
            count += 1
            if j ==-1 or self.strValue[i] == p.charAt(j):
                # 比较的字符相等或者比较主串的下一个字符
                i += 1
                j += 1
            else:
                j = next[j]
        if j == p.length():
            return i - j,count # 匹配
        else:
            return -1,count

    def display(self):
        '''打印字符串'''
        for i in range(self.curLen):
            print(self.strValue[i],end = '')
```

```
s = SqString('abcbcaabc')
p = SqString('bcaa')
i,c = s.KMP(p,0)
if i > 0:
    print('匹配')
else:
    print('不匹配')
```

2. 当稀疏矩阵中非零元素的位置或个数经常发生变化时不宜采用三元组顺序表存储结构，而应该采用链式存储结构表示。十字链表是稀疏矩阵的另一种存储结构，在十字链表中稀疏矩阵的非零元素用一个节点来表示，每个节点由 5 个域组成，如图 B.2 所示。其中，row 域存放该元素的行号，col 域存放该元素的列号，value 域存放该元素的值，right 域存放与该元素同行的下一个非零元素节点的指针，down 域存放与该元素同列的下一个非零元素节点的指针。每个非零数据元素节点既是某个行链表中的一个节点，也是某个列链表中的节点，整个稀疏矩阵构成了一个十字交叉的链表，这样的链表称为十字链表。

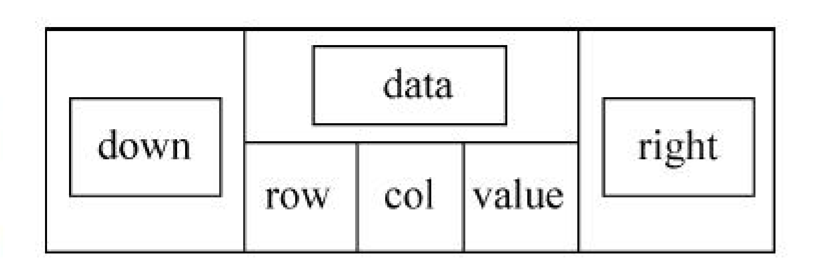


图 B.2　稀疏矩阵的 5 个域

请编写一组程序，构建 3 个类，即三元组节点类 TripleNode、十字链表存储的节点类 OLNode 和十字链表存储类 CrossList，实现十字链表的存储。当输入一组稀疏矩阵数据时能输出矩阵的非零元素的个数，并分别从行和列输出非零元素。

```
class OLNode(object):
    def __init__(self,data = None,right = None,down = None):
        self.data = data      # 三元组存储的数据包括该元素所在的行、列和数值
        self.right = right    # 行链表指针
        self.down = down      # 列链表指针

class TripleNode(object):
    def __init__(self,value,row,col):
        self.value = value
        self.row = row
        self.col = col

class CrossList(object):
    def __init__(self,rows,cols):
        self.initHeader(rows,cols)

    def initHeader(self,rows,cols):
        self.rows = rows      # 原始矩阵的行数
        self.cols = cols      # 原始矩阵的列数
        self.rhead = [None] * rows # 行指针——单纯地充当头指针,执行该列的第一个非零元素,其长度等于 rows
        self.chead = [None] * cols # 列指针——单纯地充当头指针,执行该行的第一个非零元素,所以其长度等于 cols
        self.nums = 0         # 原始矩阵中非零元素的个数
        # 初始化行的头指针
        for i in range(rows):
```

```
            self.rhead[i] = OLNode()
        # 初始化列的头指针
        for i in range(cols):
            self.chead[i] = OLNode()

    def insert(self,row,col,value):
        self.nums += 1
        data = TripleNode(value,row,col)
        newNode = OLNode(data)
        # 通过行、列头指针确定指向该新节点的指针
        t = self.rhead[row] # 找到该行的头指针
        while t.right is not None: # 找到该行的末尾
            t = t.right
        t.right = newNode   # 让该行的末尾指向该新节点
        t = self.chead[col] # 找到该列的头指针
        while t.down is not None: # 找到该列的末尾
            t = t.down
        t.down = newNode    # 让该列的末尾指向该新节点

    def printArrOfRC(self):
        print("原始矩阵 共%s行%s列,%s个非零元素" % (self.rows,self.cols,self.nums))
        print('----------------------------------------')
        print('从行来看')
        for row in range(self.rows):
            print('行%s:' % row,end='')
            t = self.rhead[row].right
            while t is not None:
                data = t.data
                print('(value:%s,row:%s,col:%s)' % (data.value,data.row,data.col),end='')
                t = t.right
            print()
        print('----------------------------------------')
        print('从列来看')
        for col in range(self.cols):
            print('列%s:' % col,end='')
            t = self.chead[col].down
            while t is not None:
                data = t.data
                print('(value:%s,row:%s,col:%s)' % (data.value,data.row,data.col),end='')
                t = t.down
            print()

    def List2CrossList(datas):
        '''根据列表格式的二维矩阵创建稀疏矩阵并返回'''
        rows = len(datas)
        cols = len(datas[0])
        cl = CrossList(rows,cols)
```

```
        for row in range(rows):
            for col in range(cols):
                value = datas[row][col]
                if value != 0:
                    cl.insert(row,col,value)
        return cl

cl = CrossList.List2CrossList(
    [
        [0,0,1,0],
        [1,0,0,4],
        [0,0,3,0],
        [1,2,0,4],
    ]
)
cl.printArrOfRC()
```

代码输出：

```
原始矩阵 共 4 行 4 列,7 个非零元素
----------------------------------------
从行来看
行 0:(value:1,row:0,col:2)
行 1:(value:1,row:1,col:0) (value:4,row:1,col:3)
行 2:(value:3,row:2,col:2)
行 3:(value:1,row:3,col:0) (value:2,row:3,col:1) (value:4,row:3,col:3)
----------------------------------------
从列来看
列 0:(value:1,row:1,col:0) (value:1,row:3,col:0)
列 1:(value:2,row:3,col:1)
列 2:(value:1,row:0,col:2) (value:3,row:2,col:2)
列 3:(value:4,row:1,col:3) (value:4,row:3,col:3)
```

第 5 章　树形结构

1. 以{1,2,3,4,5,6,7,8,9}为元素构造一棵二叉树，并输出它的先序遍历、中序遍历和后序遍历的结果。

输入：

1 2 3 4 5 6 7 8 9

输出：

先序遍历：

1 2 4 8 9 5 3 6 7

中序遍历：

8 4 9 2 5 1 6 3 7

后序遍历：

8 9 4 5 2 6 7 3 1

```
from abc import ABCMeta,abstractmethod,abstractproperty

class IQueue(metaclass = ABCMeta):
    @abstractmethod
    def clear(self):
        '''将队列置空'''
        pass
    @abstractmethod
    def isEmpty(self):
        '''判断队列是否为空'''
        pass
    @abstractmethod
    def length(self):
        '''返回队列的数据元素个数'''
        pass
    @abstractmethod
    def peek(self):
        '''返回队首元素'''
        pass
    @abstractmethod
    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        pass
    @abstractmethod
    def poll(self):
        '''将队首元素删除并返回其值'''
        pass
    @abstractmethod
    def display(self):
        '''输出队列中的所有元素'''
        pass

class Node(object):
    def __init__(self,data = None,next = None):
        self.data = data
        self.next = next

class LinkQueue(IQueue):
    def __init__(self):
        self.front = None    # 队首指针
        self.rear = None     # 队尾指针

    def clear(self):
        '''将队列置空'''
        self.front = None
        self.rear = None

    def isEmpty(self):
```

```
        '''判断队列是否为空'''
        return self.front is None

    def length(self):
        '''返回队列的数据元素个数'''
        p = self.front
        i = 0
        while p is not None:
            p = p.next
            i += 1
        return i

    def peek(self):
        '''返回队首元素'''
        if self.isEmpty():
            return None
        else:
            return self.front.data

    def offer(self,x):
        '''将数据元素 x 插入到队列成为队尾元素'''
        s = Node(x,None)
        if not self.isEmpty():
            self.rear.next = s
        else:
            self.front = s
        self.rear = s

    def poll(self):
        '''将队首元素删除并返回其值'''
        if self.isEmpty():
            return None
        p = self.front
        self.front = self.front.next
        if p == self.rear: # 删除节点为队尾节点时需要修改 rear
            self.rear = None
        return p.data

    def display(self):
        '''输出队列中的所有元素'''
        p = self.front
        while p is not None:
            print(p.data,end = ' ')
            p = p.next

class BiNode(object):
    def __init__(self,data = None,lchild = None,rchild = None):
        self.data = data
        self.lchild = lchild
        self.rchild = rchild
```

```

class BiTree(object):
    def __init__(self,root = None):
        self.root = root

    def createBiTree(order):
        q = LinkQueue()
        root = BiNode(order[0])
        bt = BiTree(root)
        q.offer(root)
        for i in range(1,len(order)):
            c = order[i]
            node = q.peek()
            if node.lchild is None:
                newNode = BiNode(c)
                node.lchild = newNode
                q.offer(newNode)
            elif node.rchild is None:
                newNode = BiNode(c)
                node.rchild = newNode
                q.offer(newNode)
                q.poll()
        return bt

    def preOrder(root):
        if root is not None:
            print(root.data,end = ' ')
            BiTree.preOrder(root.lchild)
            BiTree.preOrder(root.rchild)

    def inOrder(root):
        if root is not None:
            BiTree.inOrder(root.lchild)
            print(root.data,end = ' ')
            BiTree.inOrder(root.rchild)

    def postOrder(root):
        if root is not None:
            BiTree.postOrder(root.lchild)
            BiTree.postOrder(root.rchild)
            print(root.data,end = ' ')

bt = BiTree.createBiTree('123456789')
print("先序遍历:")
BiTree.preOrder(bt.root)
print()
print("中序遍历:")
BiTree.inOrder(bt.root)
print()
print("后序遍历:")
```

```
BiTree.postOrder(bt.root)
print()
```

代码输出：

```
先序遍历：
1 2 4 8 9 5 3 6 7
中序遍历：
8 4 9 2 5 1 6 3 7
后序遍历：
8 9 4 5 2 6 7 3 1
```

2. 二叉查找树也是一种常用的树状结构，它具有以下性质。

(1) 若它的左子树不空，则其左子树上的所有节点的值均小于它根节点的值。

(2) 若它的右子树不空，则其右子树上的所有节点的值均大于它根节点的值。

(3) 它的左、右子树也分别为二叉查找树。

我们分别定义二叉查找树的几个操作。

1) 查找操作

在二叉查找树中查找 x 的过程如下。

(1) 若二叉树是空树，则查找失败。

(2) 若 x 等于根节点的数据，则查找成功。

(3) 若 x 小于根节点的数据，则递归查找其左子树，否则递归查找其右子树。

2) 插入操作

二叉查找树 b 插入 x 的过程如下。

(1) 若 b 是空树，则直接将插入的节点作为根节点插入。

(2) 若 x 等于 b 的根节点的数据的值，则直接返回。

(3) 若 x 小于 b 的根节点的数据的值，则将 x 要插入的节点的位置改变为 b 的左子树，否则将 x 插入的节点的位置改变为 b 的右子树。

3) 删除操作

二叉查找树的删除操作（这里根据值删除，而非节点）分 3 种情况。不过在此之前应该确保根据给定的值找到了要删除的节点，如果没找到该节点则不会执行删除操作。

下面 3 种情况假设已经找到了要删除的节点。

(1) 如果节点为叶子节点（没有左、右子树），此时删除该节点不会破坏树的结构，直接删除即可，并修改其父节点指向它的引用为 null。

(2) 如果其节点只包含左子树或者右子树，此时直接删除该节点，并将其左子树或者右子树设置为其父节点的左子树或者右子树即可，此操作不会破坏树结构。

(3) 当节点的左、右子树都不空的时候，一般的删除策略是用其右子树的最小数据（容易找到）代替要删除的节点数据并递归删除该节点（此时为 null），因为右子树的最小节点不可能有左孩子，所以第二次删除较为容易。z 的左子树和右子树均不空。找到 z 的后继 y，因为 y 一定没有左子树，所以可以删除 y，让 y 的父节点成为 y 的右子树的父节点，并用 y 的值代替 z 的值，如图 B.3 所示。

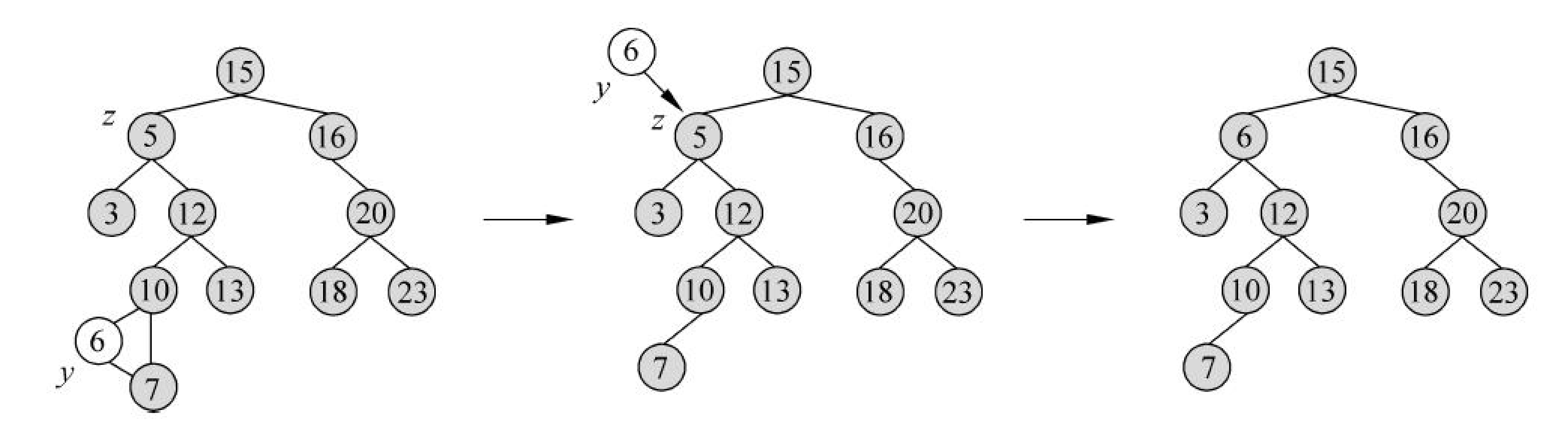


图 B.3　二叉查找树的删除操作

请用代码实现二叉查找树的上述操作。

```
class BiTreeNode(object):
    def __init__(self,key,data,lchild = None,rchild = None):
        self.key = key          # 节点关键字值
        self.data = data        # 节点的数据值
        self.lchild = lchild    # 节点的左孩子
        self.rchild = rchild    # 节点的右孩子

class BSTree(object):
    def __init__(self,root = None):
        self.root = root        # 树的根节点

    def display(self,p):
        if p is not None:
            print(p.data,end = '')
            print('(',end = '')
            self.display(p.lchild)
            print(',',end = '')
            self.display(p.rchild)
            print(')',end = '')

    def search(self,key):
        return self.searchBST(key,self.root)

    def searchBST(self,key,p):
        if p is None: # 查找树为空,查找失败
            return None
        if key == p.key: # 查找成功
            return p.data
        elif key < p.key: # 在左子树中查找
            return self.searchBST(key,p.lchild)
        else: # 在右子树中查找
            return self.searchBST(key,p.rchild)

    def insert(self,key,data):
        p = BiTreeNode(key,data) # 为元素建立节点
        if self.root is None: # 若根节点为空,建立新根节点
            self.root = p
```

```
        else:
            self.insertBST(self.root,p)

    def insertBST(self,r,p):
        if r.key<p.key: # 查找右子树
            if r.rchild is None:
                r.rchild = p
            else:
                self.insertBST(r.rchild,p)
        else: # 查找左子树
            if r.lchild is None:
                r.lchild = p
            else:
                self.insertBST(r.lchild,p)

    def remove(self,key):
        # 删除关键字为 key 的节点
        self.removeBST(key,self.root,None)

    def removeBST(self,key,p,parent):
        if p is None: # 树空,直接返回
            return
        if p.key>key: # 在左子树中删除
            self.removeBST(key,p.lchild,p)
        elif p.key<key: # 在右子树中删除
            self.removeBST(key,p.rchild,p)
        elif p.lchild is not None and p.rchild is not None: # 删除此节点,左右子树非空
            inNext = p.rchild
            while inNext.lchild is not None:
                inNext = inNext.lchild
            p.data = inNext.data
            p.key = inNext.key
            self.removeBST(p.key,p.rchild,p)
        else: # 只有一棵子树或者没有子树
            if parent is None:
                if p.lchild is not None:
                    self.root = p.lchild
                else:
                    self.root = p.rchild
                return
            if p == parent.lchild:
                if p.lchild is not None:
                    parent.lchild = p.lchild
                else:
                    parent.lchild = p.rchild
            elif p == parent.rchild:
                if p.lchild is not None:
                    parent.rchild = p.lchild
                else:
                    parent.rchild = p.rchild
```

```
bst = BSTree()
data = [15,5,3,12,10,13,6,7,16,20,18,23]
for d in data:
    bst.insert(d,d)
bst.display(bst.root)
print()
bst.remove(6)
bst.display(bst.root)
```

代码输出：

```
15(5(3(,),12(10(6(,7(,)),),13(,))),16(,20(18(,),23(,))))
15(5(3(,),12(10(7(,),),13(,))),16(,20(18(,),23(,))))
```

第 6 章　图

1. 编写一个程序,要求找出给定无向图从 *A* 点开始到所有点的最短路径。要求输出 *A* 到各个点的最短路径的距离,格式如“*A*->*A* 的最短路径为：*A* 长度：0”。

```
from abc import ABCMeta,abstractmethod,abstractproperty
import sys

class IGraph(metaclass = ABCMeta):
    @abstractmethod
    def createGraph(self):
        '''创建图'''
        pass
    @abstractmethod
    def getVNum(self):
        '''返回图中的顶点数'''
        pass
    @abstractmethod
    def getENum(self):
        '''返回图中的边数'''
        pass
    @abstractmethod
    def getVex(self,i):
        '''返回位置为 i 的顶点值'''
        pass
    @abstractmethod
    def locateVex(self,x):
        '''返回值为 x 的顶点位置'''
        pass
    @abstractmethod
    def firstAdj(self,i):
        '''返回节点的第一个邻接点'''
        pass
```

```
    @abstractmethod
    def nextAdj(self,i,j):
        '''返回相对于 j 的下一个邻接点'''
        pass

class MGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind                          # 图的种类
        self.vNum = vNum                          # 图的顶点数
        self.eNum = eNum                          # 图的边数
        self.v = v                                # 顶点列表
        self.e = e                                # 邻接矩阵

    def createUDG(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [0] * vNum ] * vNum            # 构造边集
        for i in range(eNum):
            a,b = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = self.e[n][m] = 1

    def createDG(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [0] * vNum ] * vNum            # 构造边集
        for i in range(eNum):
            a,b = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = 1

    def createUDN(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [sys.maxsize] * vNum ] * vNum  # 初始化边集
```

```
        for i in range(eNum):
            a,b,w = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = self.e[n][m] = w

    def createDN(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [sys.maxsize] * vNum ] * vNum   # 初始化边集
        for i in range(eNum):
            a,b,w = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = w

    def locateVex(self,x):
        for i in range(self.vNum):
            if self.v[i] == x:
                return i
        return -1

    def firstAdj(self,i):
        if i<0 or i>=self.vNum:
            raise Exception("第%s个顶点不存在" % i)
        for j in range(self.vNum):
            if self.e[i][j]!= 0 and self.e[i][j]< sys.maxsize:
                return j
        return -1

    def nextAdj(self,i,j):
        if j == self.vNum-1:
            return -1
        for k in range(j+1,self.vNum):
            if self.e[i][k]!= 0 and self.e[i][k]< sys.maxsize:
                return k
        return -1

class VNode(object):
    def __init__(self,data = None,firstNode = None):
        self.data = data                          # 存放节点值
        self.firstArc = firstNode                 # 第一条边

class ArcNode(object):
    def __init__(self,adjVex,value,nextArc = None):
        self.adjVex = adjVex                      # 边指向的顶点的位置
        self.value = value                        # 边的权值
        self.nextArc = nextArc                    # 指向下一条边
```

```
class ALGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind                              # 图的种类
        self.vNum = vNum                              # 图的顶点数
        self.eNum = eNum                              # 图的边数
        self.v = v                                    # 顶点列表
        self.e = e                                    # 边信息

    def createGraph(self):
        if self.kind == self.GRAPHKIND_UDG:
            self.createUDG()
        elif self.kind == self.GRAPHKIND_DG:
            self.createDG()
        elif self.kind == self.GRAPHKIND_UDN:
            self.createUDN()
        elif self.kind == self.GRAPHKIND_DN:
            self.createDN()

    def createUDG(self):
        '''创建无向图'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,1)
            self.addArc(v,u,1)

    def createDG(self):
        '''创建有向图'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,1)

    def createUDN(self):
        '''创建无向网'''
        v = self.v
```

```
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b,w = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,w)
            self.addArc(v,u,w)

    def createDN(self):
        '''创建有向网'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b,w = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,w)

    def addArc(self, i,j,value):
        '''插入边节点'''
        arc = ArcNode(j,value)
        arc.nextArc = self.v[i].firstArc
        self.v[i].firstArc = arc

    def firstAdj(self,i):
        '''查找第一个邻接点'''
        if i<0 or i>=self.vNum:
            raise Exception("第%s个节点不存在" % i)
        p = self.v[i].firstArc
        if p is not None:
            return p.adjVex
        return -1

    def nextAdj(self,i,j):
        '''返回i相对于j的下一个邻接点'''
        if i<0 or i>=self.vNum:
            raise Exception("第%s个节点不存在" % i)
        p = self.v[i].firstArc
        while p is not None:
            if p.adjVex == j:
                break
            p = p.nextArc
        if p.nextArc is not None:
            return p.nextArc.adjVex
        return -1

    def getVNum(self):
        '''返回顶点数'''
```

```
        return self.vNum

    def getENum(self):
        '''返回边数'''
        return self.eNum

    def getVex(self,i):
        '''返回第 i 个顶点的值'''
        if i<0 or i>=self.vNum:
            raise Exception("第%s个顶点不存在" % i)
        return self.v[i].data

    def locateVex(self,x):
        '''返回值为 x 的顶点的位置'''
        for i in range(self.vNum):
            if self.v[i].data == x:
                return i
        return -1

    def getArcs(self,u,v):
        '''返回顶点 u 到顶点 v 的距离'''
        if u<0 or u>=self.vNum:
            raise Exception("第%s个节点不存在" % u)
        if v<0 or v>=self.vNum:
            raise Exception("第%s个节点不存在" % v)
        p = self.v[u].firstArc
        while p is not None:
            if p.adjVex == v:
                return p.value
            p = p.nextArc
        return sys.maxsize

class CloseEdge(object):
    def __init__(self,adjVex,lowCost):
        self.adjVex = adjVex                        # 在结合 U 中的顶点的值
        self.lowCost = lowCost                      # 到集合 U 的最小距离

class ShortestPath(object):
    def Djikstra(g,v0):
        # 存放最短路径,p[v][k]表示从 v0 到 v 的最短路径中经过的第 k 个点
        p = [ ([ -1 ] * g.getVNum()) for i in range(g.getVNum()) ]
        # 存放最短路径长度
        D = [ sys.maxsize ] * g.getVNum()
        # 若已找到最短路径,finish[v]为 True
        finish = [ False ] * g.getVNum()
        v0 = g.locateVex(v0)
        for v in range(g.getVNum()):
            D[v] = g.getArcs(v0,v)
            if D[v]< sys.maxsize:
                # 从起点直接可以到达
```

```
                p[v][0] = v0
                p[v][1] = v
        p[v0][0] = v0                        # 起点本身可以直接到达
        D[v0] = 0
        finish[v0] = True
        v = -1
        for i in range(1,g.getVNum()):
            minvalue = sys.maxsize
            for w in range(g.getVNum()):
                # 找出所有最短路径中的最小值
                if not finish[w]:
                    if D[w]< minvalue:
                        v = w
                        minvalue = D[w]
            finish[v] = True
            # 更新当前的最短路径
            for w in range(g.getVNum()):
                if not finish[w] and g.getArcs(v,w)< sys.maxsize and (minvalue + g.getArcs(v,w)<D[w]):
                    D[w] = minvalue + g.getArcs(v,w)
                    for k in range(g.getVNum()):
                        p[w][k] = p[v][k]
                        if p[w][k] == -1:
                            p[w][k] = w
                            break
        dis = { g.getVex(i):D[i] for i in range(g.getVNum()) }
        # 返回到各点最短路的字典与路径矩阵
        return dis,p

    def printDjikstraPath(g,v0,p,dis):
        # v0 到各点的输出最短路径,即 p[v][0]到 p[v][j]直到 p[v][j]==-1
        u = v0
        v0 = g.locateVex(v0)
        for i in range(g.getVNum()):
            v = g.getVex(i)
            print('%s->%s 的最短路径为:' % (u,v),end='')
            if p[i][0]!=-1:
                print(g.getVex(p[v0][0]),end='')
            for k in range(1,g.getVNum()):
                if p[i][k]==-1:
                    break
                print('->%s' % g.getVex(p[i][k]),end='')
            print('长度: %s' % dis[v])


v= ['A','B','C','D','E','F']
e= [
    ('A','B',7),('A','C',5),('A','D',1),
    ('B','D',6),('B','E',3),
    ('C','D',7),('C','F',2),
```

```
    ('D','E',6),('D','F',4),
    ('E','F',7),
]
g = ALGraph(ALGraph.GRAPHKIND_UDN,len(v),len(e),v,e)
g.createGraph()
dis,p = ShortestPath.Djikstra(g,'A')
ShortestPath.printDjikstraPath(g,'A',p,dis)
```

代码输出：

```
A->A的最短路径为：A 长度：0
A->B的最短路径为：A->B 长度：7
A->C的最短路径为：A->C 长度：5
A->D的最短路径为：A->D 长度：1
A->E的最短路径为：A->D->E 长度：7
A->F的最短路径为：A->D->F 长度：5
```

2. 编写一个程序，请用 Kruskal 算法算出给定无向图的最小生成树。

输入：无

输出：生成树的每条边及生成树的权值之和。

```
from abc import ABCMeta,abstractmethod,abstractproperty
import sys

class IGraph(metaclass = ABCMeta):
    @abstractmethod
    def createGraph(self):
        '''创建图'''
        pass
    @abstractmethod
    def getVNum(self):
        '''返回图中的顶点数'''
        pass
    @abstractmethod
    def getENum(self):
        '''返回图中的边数'''
        pass
    @abstractmethod
    def getVex(self,i):
        '''返回位置为 i 的顶点值'''
        pass
    @abstractmethod
    def locateVex(self,x):
        '''返回值为 x 的顶点位置'''
        pass
    @abstractmethod
    def firstAdj(self,i):
        '''返回节点的第一个邻接点'''
        pass
```

```
    @abstractmethod
    def nextAdj(self,i,j):
        '''返回相对于 j 的下一个邻接点'''
        pass

class MGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind                          # 图的种类
        self.vNum = vNum                          # 图的顶点数
        self.eNum = eNum                          # 图的边数
        self.v = v                                # 顶点列表
        self.e = e                                # 邻接矩阵

    def createUDG(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [0] * vNum ] * vNum            # 构造边集
        for i in range(eNum):
            a,b = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = self.e[n][m] = 1

    def createDG(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [0] * vNum ] * vNum            # 构造边集
        for i in range(eNum):
            a,b = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = 1

    def createUDN(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [sys.maxsize] * vNum ] * vNum  # 初始化边集
```

```
        for i in range(eNum):
            a,b,w = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = self.e[n][m] = w

    def createDN(self,vNum,eNum,v,e):
        self.vNum = vNum
        self.eNum = eNum
        self.v = [None] * vNum                    # 构造顶点集
        for i in range(vNum):
            self.v[i] = v[i]
        self.e = [ [sys.maxsize] * vNum ] * vNum  # 初始化边集
        for i in range(eNum):
            a,b,w = e[i]
        m,n = self.locateVex(a),self.locateVex(b)
        self.e[m][n] = w

    def locateVex(self,x):
        for i in range(self.vNum):
            if self.v[i] == x:
                return i
        return -1

    def firstAdj(self,i):
        if i<0 or i>=self.vNum:
            raise Exception("第%s个顶点不存在" % i)
        for j in range(self.vNum):
            if self.e[i][j]!=0 and self.e[i][j]<sys.maxsize:
                return j
        return -1

    def nextAdj(self,i,j):
        if j==self.vNum-1:
            return -1
        for k in range(j+1,self.vNum):
            if self.e[i][k]!=0 and self.e[i][k]<sys.maxsize:
                return k
        return -1

class VNode(object):
    def __init__(self,data=None,firstNode=None):
        self.data = data                          # 存放节点值
        self.firstArc = firstNode                 # 第一条边

class ArcNode(object):
    def __init__(self,adjVex,value,nextArc=None):
        self.adjVex = adjVex                      # 边指向的顶点的位置
        self.value = value                        # 边的权值
        self.nextArc = nextArc                    # 指向下一条边

```

```
class ALGraph(IGraph):
    # 图类别静态常量
    GRAPHKIND_UDG = 'UDG'
    GRAPHKIND_DG = 'DG'
    GRAPHKIND_UDN = 'UDN'
    GRAPHKIND_DN = 'DN'

    def __init__(self,kind = None,vNum = 0,eNum = 0,v = None,e = None):
        self.kind = kind                        # 图的种类
        self.vNum = vNum                        # 图的顶点数
        self.eNum = eNum                        # 图的边数
        self.v = v                              # 顶点列表
        self.e = e                              # 边信息

    def createGraph(self):
        if self.kind == self.GRAPHKIND_UDG:
            self.createUDG()
        elif self.kind == self.GRAPHKIND_DG:
            self.createDG()
        elif self.kind == self.GRAPHKIND_UDN:
            self.createUDN()
        elif self.kind == self.GRAPHKIND_DN:
            self.createDN()

    def createUDG(self):
        '''创建无向图'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,1)
            self.addArc(v,u,1)

    def createDG(self):
        '''创建有向图'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,1)

    def createUDN(self):
        '''创建无向网'''
        v = self.v
```

```
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b,w = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,w)
            self.addArc(v,u,w)

    def createDN(self):
        '''创建有向网'''
        v = self.v
        self.v = [ None ] * self.vNum
        for i in range(self.vNum):
            self.v[i] = VNode(v[i])
        for i in range(self.eNum):
            a,b,w = self.e[i]
            u,v = self.locateVex(a),self.locateVex(b)
            self.addArc(u,v,w)

    def addArc(self, i,j,value):
        '''插入边节点'''
        arc = ArcNode(j,value)
        arc.nextArc = self.v[i].firstArc
        self.v[i].firstArc = arc

    def firstAdj(self,i):
        '''查找第一个邻接点'''
        if i < 0 or i >= self.vNum:
            raise Exception("第%s个节点不存在" % i)
        p = self.v[i].firstArc
        if p is not None:
            return p.adjVex
        return -1

    def nextAdj(self,i,j):
        '''返回i相对于j的下一个邻接点'''
        if i < 0 or i >= self.vNum:
            raise Exception("第%s个节点不存在" % i)
        p = self.v[i].firstArc
        while p is not None:
            if p.adjVex == j:
                break
            p = p.nextArc
        if p.nextArc is not None:
            return p.nextArc.adjVex
        return -1

    def getVNum(self):
        '''返回顶点数'''
```

```
            return self.vNum

    def getENum(self):
        '''返回边数'''
        return self.eNum

    def getVex(self,i):
        '''返回第 i 个顶点的值'''
        if i<0 or i>=self.vNum:
            raise Exception("第%s个顶点不存在" % i)
        return self.v[i].data

    def locateVex(self,x):
        '''返回值为 x 的顶点的位置'''
        for i in range(self.vNum):
            if self.v[i].data == x:
                return i
        return -1

    def getArcs(self,u,v):
        '''返回顶点 u 到顶点 v 的距离'''
        if u<0 or u>=self.vNum:
            raise Exception("第%s个节点不存在" % u)
        if v<0 or v>=self.vNum:
            raise Exception("第%s个节点不存在" % v)
        p = self.v[u].firstArc
        while p is not None:
            if p.adjVex == v:
                return p.value
            p = p.nextArc
        return sys.maxsize

class CloseEdge(object):
    def __init__(self,adjVex,lowCost):
        self.adjVex = adjVex                        # 在集合 U 中的顶点的值
        self.lowCost = lowCost                      # 到集合 U 的最小距离

class MiniSpanTree(object):

    def PRIM(g,u):
        '''从值为 u 的顶点出发构造最小生成树,返回由生成树边组成的二维数组'''
        tree = [[None,None] for i in range(g.getVNum()-1)]
        count = 0
        closeEdge = [None] * g.getVNum()
        k = g.locateVex(u)
        for j in range(g.getVNum()):
            if k!= j:
                closeEdge[j] = CloseEdge(u,g.getArcs(k,j))
        closeEdge[k] = CloseEdge(u,0)

```

```
        for i in range(1,g.getVNum()):
            k = MiniSpanTree.getMinMum(closeEdge)
            tree[count][0] = closeEdge[k].adjVex
            tree[count][1] = g.getVex(k)
            count += 1
            closeEdge[k].lowCost = 0
            for j in range(g.getVNum()):
                if g.getArcs(k,j)< closeEdge[j].lowCost:
                    closeEdge[j] = CloseEdge(g.getVex(k),g.getArcs(k,j))
        return tree

    def getMinMum(closeEdge):
        minvalue = sys.maxsize
        v = -1
        for i in range(len(closeEdge)):
            if closeEdge[i].lowCost!= 0 and closeEdge[i].lowCost < minvalue:
                minvalue = closeEdge[i].lowCost
                v = i
        return v

    def printMiniSpanTree(g,tree):
        length = 0
        for edge in tree:
            u,v = edge[0],edge[1]
            edgeLength = g.getArcs(g.locateVex(u),g.locateVex(v))
            length += edgeLength
            print('%s-%s:%s,'%(u,v,edgeLength),end='')
        print('总权值:%s' % length)

v= ['A','B','C','D','E','F']
e= [
    ('A','B',7),('A','C',5),('A','D',1),
    ('B','D',6),('B','E',3),
    ('C','D',7),('C','F',2),
    ('D','E',6),('D','F',4),
    ('E','F',7),
]
g= ALGraph(ALGraph.GRAPHKIND_UDN,len(v),len(e),v,e)
g.createGraph()
tree= MiniSpanTree.PRIM(g,'A')
MiniSpanTree.printMiniSpanTree(g,tree)
```

第7章　排　　序

1. 已知一个有穷整数数组,请采用冒泡排序完成从小到大的排序操作。

输入数组:

{1,2,5,4,7,6,3,0}

输出:

{0,1,2,3,4,5,6,7}

```
class RecordNode(object):
    def __init__(self,key,data):
        self.key = key                          # 关键字
        self.data = data                        # 数据元素的值

class SqList(object):
    def __init__(self,maxSize):
        self.maxSize = maxSize                  # 顺序表的最大存储空间
        self.list = [ None ] * self.maxSize     # 待排序的记录集合
        self.len = 0                            # 顺序表的长度

    def insert(self,i,x):
        # 在第 i 个位置之前插入记录 x
        if self.len == self.maxSize:
            raise Exception("顺序表已满")
        if i < 0 or i > self.len:
            raise Exception("插入位置不合理")
        for j in range(self.len,i,-1):
            self.list[j] = self.list[j-1]
        self.list[i] = x
        self.len += 1

    def display(self):
        for i in range(self.len):
            print(self.list[i].key,end = '')
        print()


    def bubbleSort(self):
        flag = True
        i = 1
        while i < self.len and flag:
            flag = False
            for j in range(self.len - i):
                if self.list[j+1].key < self.list[j].key:
                    p = self.list[j]
                    self.list[j] = self.list[j+1]
                    self.list[j+1] = p
                    flag = True
            i += 1
sl= SqList(8)
data= [1,2,5,4,7,6,3,0]
for i,x in zip(range(len(data)),data):
    sl.insert(i,RecordNode(x,x))
sl.bubbleSort()
sl.display()
```

2. 已知一个有穷整数数组,请采用快速排序完成从小到大的排序操作。

输入数组:

{1,2,5,4,7,6,3,0}

输出：

{0,1,2,3,4,5,6,7}

```
class RecordNode(object):
    def __init__(self,key,data):
        self.key = key                          # 关键字
        self.data = data                        # 数据元素的值

class SqList(object):
    def __init__(self,maxSize):
        self.maxSize = maxSize                  # 顺序表的最大存储空间
        self.list = [ None ] * self.maxSize     # 待排序的记录集合
        self.len = 0                            # 顺序表的长度

    def insert(self,i,x):
        # 在第 i 个位置之前插入记录 x
        if self.len == self.maxSize:
            raise Exception("顺序表已满")
        if i < 0 or i > self.len:
            raise Exception("插入位置不合理")
        for j in range(self.len,i,-1):
            self.list[j] = self.list[j-1]
        self.list[i] = x
        self.len += 1

    def display(self):
        for i in range(self.len):
            print(self.list[i].key,end=' ')
        print()

    def qSort(self,low,high):
        if low < high:
            p = self.Partition(low,high)
            self.qSort(low,p-1)
            self.qSort(p+1,high)


    def Partition(self,low,high):
        p = self.list[low]
        while low < high:
            while low < high and self.list[high].key > p.key:
                high -= 1
            if low < high:
                self.list[low] = self.list[high]
                low += 1
            while low < high and self.list[low].key < p.key:
                low += 1
            if low < high:
                self.list[high] = self.list[low]
```

```
                    high -= 1
            self.list[low] = p
            return low

sl = SqList(8)
data = [1,2,5,4,7,6,3,0]
for i,x in zip(range(len(data)),data):
    sl.insert(i,RecordNode(x,x))
sl.qSort(0,sl.len - 1)
sl.display()
```

3. 已知一个有穷整数数组,请采用归并排序完成从小到大的排序操作。

输入数组:

{1,2,5,4,7,6,3,0}

输出:

{0,1,2,3,4,5,6,7}

```
class RecordNode(object):
    def __init__(self,key,data):
        self.key = key                          # 关键字
        self.data = data                        # 数据元素的值

class SqList(object):
    def __init__(self,maxSize):
        self.maxSize = maxSize                  # 顺序表的最大存储空间
        self.list = [ None ] * self.maxSize     # 待排序的记录集合
        self.len = 0 # 顺序表的长度

    def insert(self,i,x):
        # 在第 i 个位置之前插入记录 x
        if self.len == self.maxSize:
            raise Exception("顺序表已满")
        if i < 0 or i > self.len:
            raise Exception("插入位置不合理")
        for j in range(self.len,i, - 1):
            self.list[j] = self.list[j - 1]
        self.list[i] = x
        self.len += 1

    def display(self):
        for i in range(self.len):
            print(self.list[i].key,end = ' ')
        print()

    def merge(self,order,a,i,k,j):
        t = i
        m = i
        n = k + 1
        while m < = k and n < = j:
```

```
            # 将具有较小关键字值的元素放入 order[]
            if a[m].key <= a[n].key:
                order[t] = a[m]
                t += 1
                m += 1
            else:
                order[t] = a[n]
                t += 1
                n += 1
        while m <= k:
            order[t] = a[m]
            t += 1
            m += 1
        while n <= j:
            order[t] = a[n]
            t += 1
            n += 1

    def mergepass(self,order,a,s,n):
        p = 0
        while p + 2 * s - 1 <= n - 1: # 两两归并长度均为 s 的有序表
            self.merge(order,a,p,p + s - 1,p + 2 * s - 1)
            p = p + 2 * s
        if p + s - 1 < n - 1: # 归并长度不等的有序表
            self.merge(order,a,p,p + s - 1,n - 1)
        else: # 将一个有序表中的元素放入 order[]中
            for i in range(p,n):
                order[i] = a[i]

    def mergeSort(self):
        s = 1 # 已排序的子序列的长度,初始值为 1
        order = [ None ] * self.len
        while s < self.len: # 归并过程
            self.mergepass(order,self.list,s,self.len)
            s = s * 2
            self.mergepass(self.list,order,s,self.len)
            s = s * 2

sl = SqList(50)
data = [1,2,5,4,7,6,3,0]
for i,x in zip(range(len(data)),data):
    sl.insert(i,RecordNode(x,x))
sl.mergeSort()
sl.display()
```

第 8 章　查　　找

1. 输入一组字符串,已知所给字符串只包含"("和")",请使用顺序查找,求出最长的合法括号子串的长度。例如所给字符串为()()()),则最长的合法括号子串为()()(),因此输

出该子串的长度 6。

输入：

()()

输出：

4

```
def longestValidParentheses(s):
    left = right = ans = 0
    length = len(s)
    for i in range(length):
        if s[i] == '(':
            left += 1
        else:
            right += 1
        if left == right:
            ans = max(ans,2 * right) # 更新长度
        elif right > left:
            left = right = 0
    left = right = 0
    for i in range(length - 1, - 1, - 1):
        if s[i] == ')':
            right += 1
        else:
            left += 1
        if left == right:
            ans = max(ans,2 * left)
        elif left > right:
            left = right = 0
    return ans

s = "()()()"
print(longestValidParentheses(s))
```

2. 编写一组程序，实现一个二叉查找树的功能，可以将给定的一个数组建立成二叉树，进行动态插入、删除关键字以及将树转换为有序列表等操作。

```
class BiTreeNode(object):
    def __init__(self,key,data,lchild = None,rchild = None):
        self.key = key # 节点关键字值
        self.data = data # 节点的数据值
        self.lchild = lchild # 节点的左孩子
        self.rchild = rchild # 节点的右孩子

class BSTree(object):
    def __init__(self,root = None):
        self.root = root # 树的根节点

    def display(self,p):
        if p is not None:
```

```
                print(p.data,end = '')
                print('(',end = '')
                self.display(p.lchild)
                print(',',end = '')
                self.display(p.rchild)
                print(')',end = '')

    def search(self,key):
        return self.searchBST(key,self.root)

    def searchBST(self,key,p):
        if p is None: # 查找树为空,查找失败
            return None
        if key == p.key: # 查找成功
            return p.data
        elif key < p.key: # 在左子树中查找
            return self.searchBST(key,p.lchild)
        else: # 在右子树中查找
            return self.searchBST(key,p.rchild)

    def insert(self,key,data):
        p = BiTreeNode(key,data) # 为元素建立节点
        if self.root is None: # 若根节点为空,建立新根节点
            self.root = p
        else:
            self.insertBST(self.root,p)

    def insertBST(self,r,p):
        if r.key < p.key: # 查找右子树
            if r.rchild is None:
                r.rchild = p
            else:
                self.insertBST(r.rchild,p)
        else: # 查找左子树
            if r.lchild is None:
                r.lchild = p
            else:
                self.insertBST(r.lchild,p)

    def remove(self,key):
        # 删除关键字为 key 的节点
        self.removeBST(key,self.root,None)

    def removeBST(self,key,p,parent):
        if p is None: # 树空,直接返回
            return
        if p.key > key: # 在左子树中删除
            self.removeBST(key,p.lchild,p)
        elif p.key < key: # 在右子树中删除
            self.removeBST(key,p.rchild,p)
```

```
            elif p.lchild is not None and p.rchild is not None: # 删除此节点,左右子树非空
                inNext = p.rchild
                while inNext.lchild is not None:
                    inNext = inNext.lchild
                p.data = inNext.data
                p.key = inNext.key
                self.removeBST(p.key,p.rchild,p)
            else: # 只有一棵子树或者没有子树
                if parent is None:
                    if p.lchild is not None:
                        self.root = p.lchild
                    else:
                        self.root = p.rchild
                    return
                if p == parent.lchild:
                    if p.lchild is not None:
                        parent.lchild = p.lchild
                    else:
                        parent.lchild = p.rchild
                elif p == parent.rchild:
                    if p.lchild is not None:
                        parent.rchild = p.lchild
                    else:
                        parent.rchild = p.rchild

    def inOrder(root,data):
        if root is not None:
            BSTree.inOrder(root.lchild,data)
            data.append(root.data)
            BSTree.inOrder(root.rchild,data)

    def toList(self):
        data = []
        BSTree.inOrder(self.root,data)
        return data

    def createBSTreee(nums):
        bst = BSTree()
        for num in nums:
            bst.insert(num,num)
        return bst

bst = BSTree.createBSTreee([15,5,3,12,10,13,6,7,16,20,18,23])
bst.display(bst.root)
print()
print(bst.toList())
bst.insert(11,11)
bst.display(bst.root)
print()
print(bst.toList())
```

```
bst.remove(13)
bst.display(bst.root)
print()
print(bst.toList())
```

代码输出：

```
15(5(3(,),12(10(6(,7(,)),),13(,))),16(,20(18(,),23(,))))
[3, 5, 6, 7, 10, 12, 13, 15, 16, 18, 20, 23]
15(5(3(,),12(10(6(,7(,)),11(,)),13(,))),16(,20(18(,),23(,))))
[3, 5, 6, 7, 10, 11, 12, 13, 15, 16, 18, 20, 23]
15(5(3(,),12(10(6(,7(,)),11(,)),)),16(,20(18(,),23(,))))
[3, 5, 6, 7, 10, 11, 12, 15, 16, 18, 20, 23]
```

参 考 文 献

[1] 叶核亚. 数据结构(Java 版)[M]. 北京：电子工业出版社，2015.
[2] 袁开友，郑孝宗. 数据结构(Java 应用案例教程)[M]. 重庆：重庆大学出版社，2014.
[3] 刘小晶，杜选. 数据结构(Java 语言描述)[M]. 北京：清华大学出版社，2011.
[4] 张强，赵莹，武岩. Java 与数据结构的应用[M]. 北京：北京邮电大学出版社，2015.
[5] 徐孝凯. 数据结构教程(Java 语言描述)[M]. 北京：清华大学出版社，2010.
[6] Mark Allen Weiss. 数据结构与问题求解(Java 语言版)[M]. 4 版. 北京：清华大学出版社，2011.
[7] 丁海军. 数据结构(Java 语言描述)[M]. 北京：电子工业出版社，2015.
[8] 雷军环，吴名星. 数据结构(Java 语言版)[M]. 北京：清华大学出版社，2015.
[9] 王学军. 数据结构(Java 语言版)[M]. 北京：人民邮电出版社，2008.
[10] John Lewis，Joseph Chase. Java 软件结构与数据结构[M]. 3 版. 北京：清华大学出版社，2009.
[11] Mark Allen Weiss. 数据结构与算法分析(Java 语言描述)[M]. 冯舜玺，译. 2 版. 北京：机械工业出版社，2015.
[12] Lafore. R. Java 数据结构和算法[M]. 2 版. 北京：中国电力出版社，2007.
[13] 车战斌，李占波. 面向对象的数据结构(Java 版)[M]. 郑州：河南科学技术出版社，2008.
[14] 蔡明志. 数据结构(Java 版)[M]. 北京：中国铁道出版社，2006.
[15] Allen B Downey. 像计算机科学家一样思考 Python [M]. 北京：人民邮电出版社，2013.
[16] Mark Lutz. Python 学习手册[M]. 4 版. 北京：机械工业出版社，2011.
[17] Alex Martelli，Anna Ravenscroft，David Ascher. Python Cookbook[M]. 北京：人民邮电出版社，2010.
[18] Kenneth A. Lambert. 数据结构 Python 语言描述[M]. 李军，译. 北京：人民邮电出版社，2017.
[19] 裘宗燕. 数据结构与算法：Python 语言描述[M]. 北京：机械工业出版社，2016.